Mike Zerkus, John Lusher, & Jonathan Ward

USB Primer
Practical Design Guide

Part 1 of 4

Before getting into the nitty-gritty of working on Universal Serial Bus projects, you need to know the basics. But Mike, John, and Jon offer more than an intro to USB. Their demo gets you ready to work on your own.

Universal serial bus (USB) promises to be the next major advance in PC functionality, completing the PC's transition to a plug-and-play system. But, for all its possibilities, USB is bit of a mystery.

For the average engineer with an idea for a USB product or who has been commanded to convert an existing system, the journey to enlightenment can be an arduous struggle. Rather than merely providing information on USB, we want to show you how to get your USB device up and running.

As a high-speed bus for connecting external devices to the PC, USB is the next step for external peripherals on Windows and Macintosh computers. By allowing hot-plug installation, reconfiguration becomes less of a hassle.

USB enables 127 devices to be on the bus simultaneously. This arrangement solves the problem of limited serial ports.

USB operates at 12 Mbps (there is a low-speed mode of 1.5 Mbps for some devices), and it supports isochronous and asynchronous data transfers. Because USB devices can be bus powered, the transformer ganglion behind the computer can be reduced.

PC users now have a simple user-friendly peripheral bus that supports up to 127 devices and that can be installed without configuring or altering their current system. Gone are the days of figuring out which interrupt settings and I/O addresses were available and altering the device's settings to fit the available resources.

With USB, you just plug the device into the port. The OS takes care of the rest. There are no jumpers, power packs, powerdowns, resets, or taking the case off. The PC automatically installs the appropriate driver and configures the device as needed.

HOW DOES USB WORK?

RS-232 serial communication with UARTs, transfer rates, stop bits, and so on traces its heritage back to mechanical devices in the days of teletype. In the heyday of TTY, you could repair a UART with a wrench. Adjusting the transfer rate was more like tuning a car than working on electronics.

USB doesn't represent an electronic analog of a mechanical system. In a USB system, the line between hardware and software function is blurred. USB exploits the full potential of a computerized communication system.

The two sides of a USB system are the device and the host. The device side consists of the USB device (e.g., modem or printer), which usually contains a USB microcontroller (e.g., the Intel '930) and the code to properly initiate USB communication to the host. The host side is the PC running an OS that supports USB. The device and host communicate over the USB cable.

USB devices can be self-powered or bus powered, so they can be produced without including a bulky wall-mount transformer. The device gets its power from the host computer or USB hub.

BUS TOPOLOGY

USB uses a tiered/star bus topology in which each device plugs into a hub. The hub is a traffic cop that enforces the low-level rules of the bus. Figure 1 shows the physical arrangement of a USB system. For the most part, hubs are transparent.

Classes are the device categories that share common I/O requirements. In USB there are currently 11 classes: common class, audio, communications, hub, human interface device (HID), image, monitor, physical interface device (PID), power, printer, and storage.

Classes introduce a set of standard drivers native to the OS (Windows 98) and enable you to use them as is, write your own driver, or have a mini-driver.

PACKETS

A packet is a combination of special fields. All packets begin with the Sync field to synchronize the phase-locked loop and have a packet identifier (PID) that helps USB devices determine the kind of packet being sent. The packet is followed by address information, a frame number, or data. There are four types of packets; each has several subtypes.

The first packet type—the token, shown in Figure 2a—is a 24-bit data

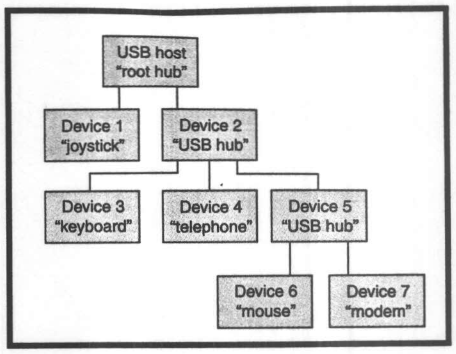


Figure 1—*This diagram shows the physical arrangement of a USB system.*

packet that represents what is happening over the bus. The first eight bits represent the packet identifier. The next seven bits are the address of the device that the host is communicating with. The next four bits are the endpoint address, which is where the data is going in the device. And, the last five bits are the CRC to check the token for errors.

There are four types of tokens—In, Out, Start of Frame (SOF), and Setup. Check the glossary of terms in Design Forum for more details. An SOF packet is illustrated in Figure 2b.

As you see in Figure 2c, data packets contain PIDs for further data error-checking. Data packets alternate between DATA0 and DATA1. The only exception to this format is the Setup packet, which always uses the DATA0 packet.

Data packets have a format of the DATA0/1 PID followed by the data, which ranges in length from 0 to 1023 bytes. The packet is checked with a 16-bit CRC field.

The handshake packet is shown in Figure 2d. These packets inform the sender of the data as to

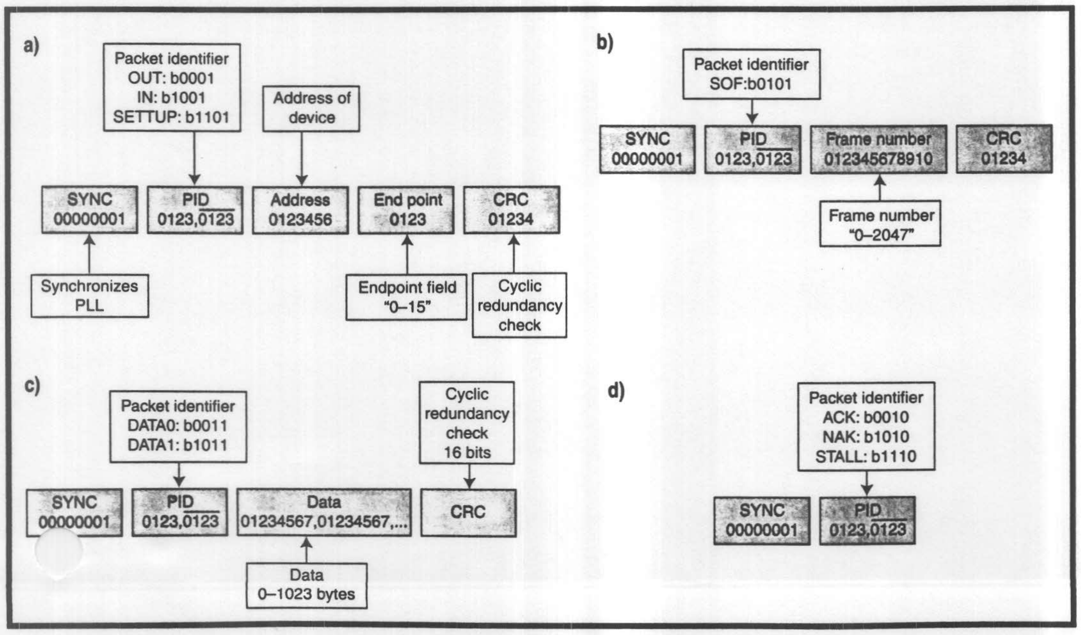
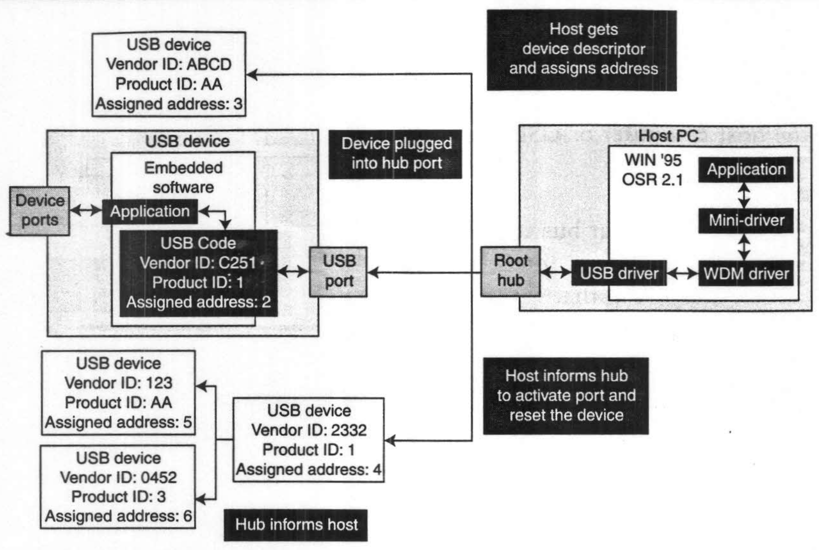


Figure 2—*These diagrams show four different types of packets: token* (**a**), *SOF* (**b**), *data* (**c**), *and handshake* (**d**).



1. Device is attached to a hub port.
2. Hub notifies host of change-of-port status when polled for status updates.
3. Host queries the hub as to the change in port status.
4. Host issues a port enable and reset command for the port.
5. Device now in powered state and replies to the default address.
6. Host retrieves the device descriptor and determines the maximum packet size for endpoint zero. It also retrieves the VID and PID to know what device driver to use with the device.
7. Host assigns unique address to USB device.
8. Host retrieves the configuration descriptors. Based on available power and bandwidth, the host assigns a configuration to the device.
9. The device is now addressed, configured, and ready to use.

Figure 3—*This diagram and the accompanying list illustrate the enumeration of a USB device.*

the condition of the received data packet. Handshake packets are ACK, NAK, and STALL.

The special preamble packet establishes low-speed communication on the bus. This token is sent full speed to the hubs, and the hubs then enable their low-speed outputs.

DESCRIPTORS

The descriptor includes general information about the device. The Vendor ID and Product ID fields play the key role in the enumeration of the device. The descriptor also informs the host about the number of configurations of the device.

Configuration descriptors tell the host the number of interfaces, the device's power requirements, and its attributes. Interface descriptors are the number of endpoints and what class they belong to as well as the interface protocol.

Endpoint descriptors describe the direction and attributes of the endpoints belonging to a specific interface, including the address of endpoint, direction of endpoint, attribute of endpoint, and maximum packet size.

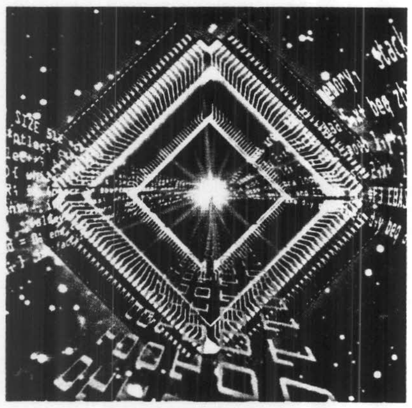
DATA TRANSFERS

A transfer or transaction consists of a number of packets moving back and forth along the bus between the host and a device. There are four types of data transfers in a USB system:

- control—controls the bus, bidirectional, setup, data, status
- bulk—asynchronous data, bidirectional, CRC
- isochronous—time-critical data, no CRC, unidirectional, up to 1023 bytes per frame, guaranteed bandwidth per frame
- interrupt—receives data at timed intervals, input only, 1–255-ms intervals

Enumeration is the bus-configuration process, which takes place anytime the bus is started or a device is plugged into or unplugged from the bus. This process is shown in Figure 3.

You always believed there were more intelligent embedded tools out there.



You were right.

People doubted their existence – yet you continued to search – and now you've found them.

COSMIC C compilers are fast, efficient, reliable, and produce the tightest object code available. Cosmic Software's embedded development tools offer portability for a complete line of microcontrollers. All toolkits include IDEA, our intuitive IDE that provides everything you need in a single, seamless Windows framework.

Add ZAP, our non-intrusive source-level debuggers and minimize your test cycle too. Want proof of their existence?

Download a free evaluation copy of our development tools at www.cosmic-software.com or call Cosmic today.

Cosmic supports the Motorola family of microcontrollers: **68HC05, 68HC08, 68HC11, 68HC12, 68HC16, 68300 and STMicroelectronics' ST7 Family.**

COSMIC Software

E-mail: c-tools@cosmic-us.com
www.cosmic-software.com

Phone:	US	**781 932-2556**
	France	**33 1 4399 5390**
	UK	**44 01256 843400**
	Germany	**49 0711 4204062**
	Sweden	**46 31704 3920**

DISTRIBUTOR for the Design99 Contest

The whole USB system is not provided by any one vendor. The OS provides some parts; other parts come from third parties and the developer.

For the next few years, most USB development projects will have to function with both Windows 95 and 98. There are some key items to be aware of when using USB with Windows 95.

Windows 95 doesn't have native USB support. You must have OSR 2.1 build 1214 or better installed on the system.

Windows 95 also has some minor bugs. One such bug is when the USB device has no alternate settings. If this occurs, Windows 95 freezes up when the device is unplugged.

Windows 98 handles USB right out of the box and resolves the above-mentioned bug. It also has a program to assist in developing USB peripherals. `usbview.exe` enables you to monitor the activity on the USB bus as well as get the device descriptors.

END-TO-END EXAMPLE

As promised, here's an example of how to get a USB device working. Using a commercially available evaluation board, the goal of this system is to blink LEDs on the eval board from the PC and to blink indicators on the PC screen from the eval board.

For this project, you need the Anchor Chips EZ-USB evaluation kit V.C or better, the USB specification, Windows PC with USB support, Windows 95 (OSR 2.1 build 1214 or better) or Windows 98, and Visual C++ or Visual Basic (V.5.0 or better). To produce drivers, you need Windows 98 and the Windows 98 DDK.

We hooked up Port A of the Anchor Chips device to a switch/LED circuit and created a DLL using the driver's `IOCTL` functions. A VB program calls the DLL and gets the data from the driver. Basically, VB requests device descriptors by calling the DLL (passes an empty pointer to buffer) and the DLL calls the driver using `IOCTL_Ezusb_GET_DEVICE_DESCRIPTOR`.

Data is passed to a buffer, and the buffer is filled and returned to VB. The driver calls the `USBD.SYS` driver to communicate with the device and OS.

HARDWARE TESTING

When you get your development board, you want to make sure the hardware works. First, install the software, which puts the EZ-USB driver into the Windows system and the install information (`INF`) file into the `INF` directory.

The `INF` file informs Windows as to what driver to load for the particular vendor ID (VID) and product ID (PID) combination. The USB Implementers Forum provides the VID; you assign the PID.

Once the software is installed, plug in the USB device with the included USB cable. It's impossible to hook up the cable backwards because the cable has two different connectors (A and B).

When the device is connected, the red light lights up, signaling that the board has power. Windows informs you that it has found new hardware, finds the appropriate `INF` file, and installs the driver for the new device.

All information concerning driver and VID/PID combinations are in the Windows system registry. If, during

LOW COST... LOW POWER...

LOW RISC!

OEM (1K) PRICE **$1.99**
EVAL KIT 7.00

A LOWER COST, FASTER, EASIER TO PROGRAM SINGLE CHIP COMPUTER

COMPARE:	16C54	MV1200
OEM (1K) PRICE	$2.57	$1.99
RS232 PROGRAM DOWNLOAD	NO	YES
SINGLE CHIP OPERATION	NO	YES
BUILT-IN BASIC	NO	YES
EEPROM DATA MEMORY	NONE	64
PROGRAM MEMORY	768 OTP	1K FLASH
MATH REGISTERS	1	32
MAX INSTRUCTIONS / SEC	5M	20M
MAX COUNTER BITS	16	18
INPUT / OUTPUT BITS	12	15
A TO D COMPARATOR	NO	YES
HARDWARE INTERRUPTS	NONE	3

PINOUT:

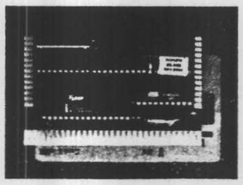
RESET	1	20	VCC
PD0	2	19	PB7
PD1	3	18	PB6
XO	4	17	PB5
XI	5	16	PB4
PD2/INT	6	15	PB3
PD3	7	14	PB2
PD4/TMR	8	13	PB1/AD1
PD5	9	12	PB0/AD0
GND	10	11	PD6

- LONGER WORD INSTRUCTION - FRIENDLY SYMMETRIC ARCHITECTURE -

CREDIT CARD COMPUTER

OEM (1K) PRICE **$21**
EVAL KIT $75.00

- -RS232/SPI program download
- -256k NV mem., expand. to 8meg
- -LCD / keyboard interface
- -ISA Bus (modem,memory,etc.)
- -RS232/485 serial to 115.2kbaud
- -Use assembly or built in BASIC
- -100 microamp OPERATION

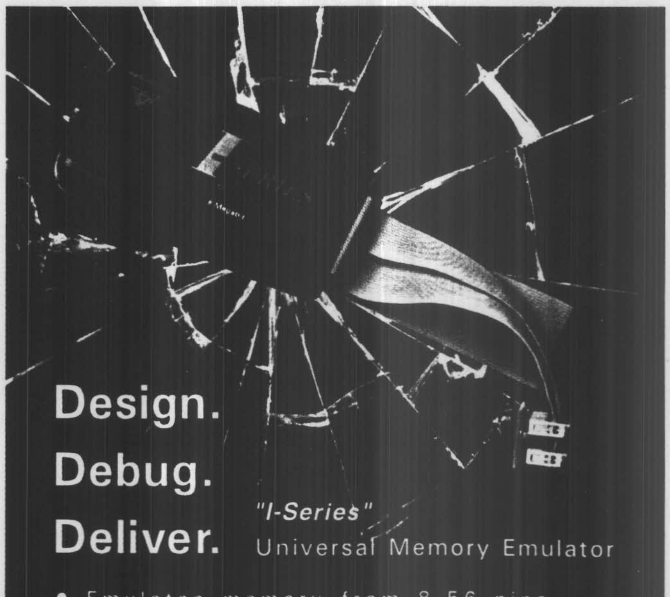


www.mvsweb.com or www.star.net/people/~mvs

MVS BOX 850
MERRIMACK,NH 03054
(508) 792-9507

MVS

5yr Limited Warranty
Free Shipping
Hrs: Mon-Fri 10-6 EST



Design.
Debug.
Deliver.

"I-Series"
Universal Memory Emulator

- Emulates memory from 8-56 pins.
- EPROM, SRAM, DRAM and more.
- Full Flash support. • 32Mbit capacity.
- Dual 8 or 16 bit. • Complex triggers.
- Advanced debugging features.

SCANLON DESIGN INC.

Tel (902)425-3938 Fax(902)425-4098
Sales & Info (800)352-9770
www.scanlondesign.com

Listing 1—*This Visual Basic code calls a DLL to get the device descriptor.*

```
Private Type UnsignedInt         ' Type define to overcome VB's limitation
                                 ' in not having unsigned 16-bit numbers
  lobyte As Byte
  hibyte As Byte
End Type

Private Type LongData
  Number As Long
End Type

Private Type USB_DD              ' Type define for USB Device Descriptor
   Descriptor_Length As Byte
   Descriptor_Type As Byte
   Spec_Release As UnsignedInt
   Device_Class As Byte
   Device_SubClass As Byte
   Device_Protocol As Byte
   Max_Packet_Size As Byte
   Vendor_ID As UnsignedInt
   Product_ID As UnsignedInt
   Device_Release As UnsignedInt
   Manufacturer As Byte
   Product As Byte
   Serial_Number As Byte
   Number_Configurations As Byte
End Type

'DLL functions used to communicate with USB device
Private Declare Function ReadBulkByte Lib "lusher_USB.dll" (ByRef InByte As Byte,
   ByVal PipeNumber As Byte, ByVal DeviceDriver As String) As Integer
Private Declare Function WriteBulkByte Lib "lusher_USB.dll" (ByVal OutByte As
   Byte, ByVal PipeNumber As Byte, ByVal DeviceDriver As String) As IntegerPrivate
Declare Function GetDeviceDescriptor Lib "lusher_USB.dll" (ByRef DevDes As USB_DD,
   ByVal DeviceDriver As String) As Integer
' get device descriptor and parse it into appropriate fields
Private Sub Get_USB_Device_Descriptor()
   ' Gets device descriptor from the USB device as well as verify
   '    that USB is communicating correctly and that correct
   '    source code is running
   Dim CheckData As Byte
   Dim ProdID As LongData
   Dim VendID As LongData
   Dim SpecRel As LongData
   Dim DevRel As LongData
   Dim USB_Device_Descriptor As USB_DD
   Dim Result As Integer

   Result = GetDeviceDescriptor(USB_Device_Descriptor, "\\.\ezusb-0")

   ' If all transactions met with success, then set up screen and
   '    allow user interaction
   ' Else alert user to the fact that no USB device is present and
   '    do not allow user interaction
   If Result = 0 Then
      Call USBError          ' Informs user of error and set form attributes
                             '    to offline mode
      Exit Sub
   End If

   Status.Caption = "USB Device Connected"
   Status.ForeColor = &HFF00&
   LSet ProdID = USB_Device_Descriptor.Product_ID
   LSet VendID = USB_Device_Descriptor.Vendor_ID
   LSet SpecRel = USB_Device_Descriptor.Spec_Release
   LSet DevRel = USB_Device_Descriptor.Device_Release
   DesForm.Type.Caption = USB_Device_Descriptor.Descriptor_Type
   DesForm.Spec.Caption = SpecRel.Number
   DesForm.Class.Caption = USB_Device_Descriptor.Device_Class
   DesForm.SubClass.Caption = USB_Device_Descriptor.Device_SubClass
   DesForm.Protocol.Caption = USB_Device_Descriptor.Device_Protocol
   DesForm.PacketSize.Caption =
      USB_Device_Descriptor.Max_PAacket_Size
   DesForm.VendorID.Caption = VendID.Number
   DesForm.ProductID.Caption = ProdID.Number
   DesForm.DevRel.Caption = DevRel.Number
   ProductID.Caption = ProdID.Number
   VendorID.Caption = VendID.Number
End Sub

' Example calls to read and write functions
Result = WriteBulkByte(Data_Out, 0, "\\.\ezusb-0")
Result = ReadBulkByte(Data_In, 7, "\\.\ezusb-0")
```

Flashlite 386Ex With New Low-Cost Multi-I/O Board

386Ex DOS Systems
STRETCH
Your I/O Budget

Flashlite 386Ex Multi-I/O Development Kit $429

Includes:
- Flashlite 386Ex: 512K Ram/512K Flash
- Multi-I/O:
  4 A/D, 2 D/A, 4 Relay Drivers, 1 UART
- Borland C/C++ 4.52
- Cable Kit
- A/C Adapter 110V
- Software Drivers
- Manuals with Schematics

Multi-I/O
Start at:
$99 qty 1
- 4 or 8 Channel 12 Bit A/D
- 2 or 4 Channel 12 Bit D/A
- 4 or 8 1A Relay Drivers
- 0 or 1 or 2 High Speed UARTS
- Software drivers included for C, QuickBasic & Assembly.
- Expands easily for demanding applications.

Flashlite 386Ex
Start at:
$219 qty 1 ***$149 qty 100***
- 25MHz
- 512K Flash
- 256 or 512K Ram
- Socket to Add Disk Storage
- 30 Parallel I/O lines
- 2 PC Compatible Serial Ports
- Clock Calendar & Watchdog
- LCD & Keypad Drivers

Also Available
Relay I/O ***$139 qty 1***
- 8 10A Relays
- 8 Opto-Isolated Inputs

Call 530-297-6073
Fax 530-297-6074
www.jkmicro.com

JK *microsystems*

Listing 2—*This example DLL handles calling the EZ-USB driver.*

```
// ReadBulkData(BYTE *OutBuffer, BYTE NumberOfBytes, BYTE PipeNumber,
//   LPSTR DeviceDriver)
// Function reads data from specific pipe over USB port for device in question
int _stdcall ReadBulkData(BYTE *OutBuffer, BYTE NumberOfBytes, BYTE PipeNumber,
  LPSTR DeviceDriver)
{
  HANDLE hUSB_DeviceHandle; // Declare variables
  DWORD  nBytes = 0;
  BOOL bResult;
  BULK_TRANSFER_CONTROL bulkControl;
  BYTE Input[64];
  BYTE index;

  bulkControl.pipeNum = (ULONG)PipeNumber;
  if (NumberOfBytes > 64 || NumberOfBytes < 1)
  // Limit ammount of transfer to 64 bytes maximum
  // If greater than 64 or less than 1 then return an error
  {
     return 0;
  }
  // Get handle to USB device in question
  hUSB_DeviceHandle = CreateFile(DeviceDriver, GENERIC_WRITE,
    FILE_SHARE_WRITE, NULL, OPEN_EXISTING, 0, NULL);
  if(hUSB_DeviceHandle == INVALID_HANDLE_VALUE)
  {
  return 0;         // If not a good handle then abort!
}

// Else it is a good handle; read data to USB pipe by calling system driver
  bResult = DeviceIoControl(hUSB_DeviceHandle,IOCTL_EZUSB_BULK_READ, &bulkControl,
    sizeof(BULK_TRANSFER_CONTROL), &Input[0], NumberOfBytes, &nBytes, NULL);
  CloseHandle(hUSB_DeviceHandle);  // Close handle
  // Fill result with that of input array
  for (index = 0; index < NumberOfBytes; index++)
  {
     *OutBuffer = Input[index];
     OutBuffer++;
  }
  return (int)bResult;   // Return our result: success or failure
}

// WriteBulkData(BYTE *InBuffer, BYTE PipeNumber, LPSTR DeviceDriver)
// Function writes data from specific pipe over USB port for device in question

int _stdcall WriteBulkData(BYTE *InBuffer, BYTE NumberOfBytes, BYTE PipeNumber,
  LPSTR DeviceDriver)
{
  HANDLE hUSB_DeviceHandle;    // Declare variables
  DWORD  nBytes = 0;
  BOOL bResult;
  BULK_TRANSFER_CONTROL bulkControl;
  BYTE Output[64];
  BYTE index;

  bulkControl.pipeNum = (ULONG)PipeNumber;
  // Limit ammount of transfer to 64 bytes maximum
  // If greater than 64 or less than 1, return an error
  if (NumberOfBytes > 64 || NumberOfBytes < 1)
  {
     return 0;
  }
  // Fill output array with that of input buffer
  for (index = 0; index < NumberOfBytes; index++)
  {
     Output[index] = *InBuffer;
     InBuffer++;
  }
  // Get handle to USB device in question
  hUSB_DeviceHandle = CreateFile(DeviceDriver, GENERIC_WRITE,
      FILE_SHARE_WRITE, NULL, OPEN_EXISTING, 0, NULL);
  if(hUSB_DeviceHandle == INVALID_HANDLE_VALUE)
  {
     return 0;         // If not a good handle, abort!
  }
  // Else it is a good handle; write data from USB pipe by calling system driver
  bResult = DeviceIoControl(hUSB_DeviceHandle,IOCTL_EZUB_BULK_WRITE,&bulkControl,
    sizeof(BULK_TRANSFER_CONTROL), &Output[0], NumberOfBytes, &nBytes, NULL);
  CloseHandle(hUSB_DeviceHandle);  // Close handle
  return (int)bResult;   // Return our result: success or failure
}
```

development, you need to delete these entries, they are located at HKEY_ LOCAL_MACHINE\Enum\USB and HKEY_ LOCAL_MACHINE\System\Current-ControlSet\Services\Class\USB.

You can use regedit.exe to edit and browse the registry entries on your computer. You can now unplug and replug the device as much as you need. Until you delete the registry entries, Windows remembers what driver to load and doesn't inform you of any hardware detection again. This step is called enumeration.

Enumeration is when the OS recognizes that there is new hardware on the bus and determines its particular needs. The appropriate driver is then loaded and it gives the device a unique address. Enumeration takes place each time you plug a device on the bus and on bootup of Windows.

At this point you should try the software package that comes with the evaluation kit. EZ-USB Control Panel lets you get the descriptors from the USB device, download firmware to the device, and run the Keil debugger.

THE GOAL

Our goal is to build a demo of a working USB device. The concept is that a user application sends data to a USB device and vice versa. Our USB device is the Anchor Chips development board running firmware we created.

On the host side, we have an application made using Visual Basic. The program communicates with the device via the general-purpose driver (GPD) from Anchor Chips and a DLL we created to implement a bridge between the user interface (VB) and the GPD.

The user interface is an experiment board with four DIP switches and four LEDs. The user selects a four-digit binary combination that appears on the LEDs and vice versa for the DIP switches.

In our VB program, we call functions in a DLL that communicates with the GPD. Listing 1 shows how to call the DLL that communicates with the USB device. It also shows how to parse up the device descriptor and read and write a byte from the USB device.

A DLL was written to communicate between the Visual Basic program and

The DLL does the necessary communicating with the system driver and if there is an error, responds to the calling application with an error status. This arrangement provides an easy-to-use interface to the GPD.

FIRMWARE

For your USB microcontroller, we recommend you have the full version of the C compiler because the example files may exceed the evaluation limit of most evaluation-level compilers. Most of the code needed to communicate with the host is already written. Just fill in your peripheral and I/O code.

The development kit has two firmware files called `PERIPH.C` and `FW.C`. These files (supplied by Anchor Chips) contain the framework for the whole 8051-based USB control code. The `PERIPH.C` source file contains the polling loop code segment, as well as the endpoint interrupts for communicating with the host. You merely write your peripheral code in the poll loop.

When data is to be exported, a set of ISRs is called (seven in and seven out). These are the endpoints of the communication pipes. In the initialization section of the code, you need to set the direction of the port pins. For our example, port A is used. The upper nibble is input and the lower nibble is output. Listing 3 shows example routines from `PERIPH.C`.

In USB the host initiates all communications. If the device has something to tell the host, it must place the data into an output array (IN1BUF[0]).

After the firmware is finished, it must download to the chip because Anchor Chips' USB paradigm calls for the device-side application code to be transferred on startup of the processor. There are two methods for downloading the firmware—B0 load and B2 load. We describe B0 here.

The micro is basically a state machine that does simple USB tasks without 8051 code. Using the B0 protocol, the firmware is sent over the USB to the chip and an external EEPROM contains the device descriptor (VID and PID). This information tells Windows to load a driver.

The driver was made using the `ezloader.sys` driver source file, which lets you implement your firmware as part of the driver. The device is enumerated to download the firmware to the micro's RAM.

The micro reenumerates and reports a new device descriptor. We used the same VID but different PIDs (x8000 for download, x8001 for device). The new VID/PID combination tells Windows to load the real driver (`EZUSB.SYS`).

An `INF` file tells Windows which VID/PID combination goes with each driver. Listing 4 is a typical `INF` file entry for the VID/PID combo. Our VID is 0x06E5, and the PIDs are 0x8000 and 0x8001. The sample `INF` file tells Windows which drivers to load according to the VID and PID information that the system retrieves from the device.

READY TO GO

Basically, you treat most of the firmware code as if you were in regular 8051 development, except that the code resides in the `POLL` loop, not `MAIN`. There are ample instructions in the kit manuals. The GPD is well documented, and their program handles the rest. ■

Mike Zerkus has 15 years of experience working on devices and inventions ranging from space devices to consumer products. Mike is the president of CM Research, a development company that specializes in bringing products from concept through prototype to production. You may reach him at mzerkus@cmresearch.com.

John Lusher is an electrical engineer and has been involved with USB development for the last two years. You may reach him at jlusher@lushertech.com.

Jonathan Ward is president of Keil Software and has been involved in the design, implementation, and documentation of embedded systems since the early 1980s. You may reach him via (972) 735-8052.

RESOURCES

A glossary of USB terms, a checklist for building a USB device, and a list of USB suppliers are available online in Design Forum in May.

Finally, Standard RS-485 Network Software

With Cimetrics' 9-Bit µLAN you can link together up to 250 of the most popular 8- and 16-bit microcontrollers (8051, 80C196, 80C186EB/EC, 68HC11, 68HC16, 68332, PIC16C74).

The 9-Bit µLAN is:

- ***Fast*** — A high speed (62.5k baud) multidrop master/ slave RS-485 network
- ***Flexible*** — Compatible with your microcontrollers
- ***Reliable*** — Robust 16-bit CRC and sequence number error checking
- ***Efficient*** — Low microcontroller resource requirements (uses your chip's built-in serial port)
- ***Friendly*** — Simple-to-use C and assembly language software libraries, with demonstration programs
- ***Complete*** — Includes network software, network monitor, and RS-485 hardware
- ***Standard*** — The 9-Bit µLAN is an asynchronous adaptation of IEEE 1118

Microcontroller or PC Master
Intel 8051 HVAC/R Lighting Control
Intel 80C186EB/EC Point-of-Sale
Intel 80C196 Transportation
Microchip PIC16C74 Fire/Security
Motorola 68HC11 Process Control Telecommunication
Motorola 68HC16 Medical Equipment
Motorola 68332 Laboratory Data Acquisition
RS-485 MULTIDROP NETWORK

Cimetrics TECHNOLOGY

55 Temple Place • Boston, MA 02111-1300 • Ph 617.350.7550 • Fx 617.350.7552

LOW COST

Dominos are rugged, miniature encapsulated controllers that combine lots of analog and digital I/O with a fast control-oriented floating-point BASIC to provide a one-stop computation and control solution for cost-sensitive control tasks. Used stand alone or connected via RS-232/RS-485, Dominos are true plug-and-go control.

Domino 1 features:

- Full floating-point ROMed BASIC
- 32-KB SRAM and 32-KB EEPROM
- 12 bits of parallel I/O
- 2 PWM outputs
- I^2C bus
- 2-channel 12-bit ADC
- Serial port: 19.2-kbps RS-232A, RS-422, or RS-485
- +5 V @ 15 mA

Domino 2 has:

- **all of the above plus**
- 16 more bits of high-current parallel I/O
- Hardware clock/calendar
- Wide-range power operation
- Hardware PWM output

$99 to $139

Visit our Web site for complete datasheets

www.micromint.com

To Order Call:

1-800-635-3355

4 Park St - Vernon CT 06066

860-871-6170

```
  EZUSB_IRQ_CLEAR();              // Clear the IRQ
  OUT07IRQ = bmEP1;
}
```

the GPD. The DLL gets a device handle to the device driver in question.

We want to communicate to the first instance of the device driver (i.e., the first device to use this driver). If we get a valid handle, we can communicate to it. Otherwise, the device isn't on the bus and the driver isn't loaded. We communicate to the driver via `DeviceIOControl`. This function passes data to and from the device driver and it returns success or failure.

Listing 2 shows how a DLL can be used to communicate with the GPD.

Listing 4—*This code shows you an example* `INF` *file.*

```
;FILE:   EXAMPLE.INF

[Version]
signature="$CHICAGO$"
Class=USB
Provider=%Exanoke%
LayoutFile=LAYOUT.INF

[Manufacturer]
%Example%=Example

[PreCopySection]
HKR,,NoSetupUI,,1

[DestinationDirs]
DefaultDestDir=11

[LusherTech]
;
%USB\VID_06E5&PID_8000.DeviceDesc%=FIRMWARE, USB\VID_06E5&PID_8000
%USB\VID_06E5&PID_8001.DeviceDesc%=USBDEV01, USB\VID_06E5&PID_8001

[ControlFlags]
ExcludeFromSelect=*          //removes all devices from device installer list

[FIRMWARE]
AddReg=FIRMWARE.AddReg

[FIRMWARE.AddReg]
HKR,,DevLoader,,*ntkern
HKR,,NTMPDriver,,firmdown.sys

[USBDEV01]
AddReg=USBDEV01.AddReg

[USBDEV01.AddReg]
HKR,,DevLoader,,*ntkern
HKR,,NTMPDriver,,ezusb.sys

[Strings]
Example="Example USB"
USB\VID_06E5&PID_8000.DeviceDesc="USB Firmware Download"
USB\VID_06E5&PID_8001.DeviceDesc="USB Actual Device"
```

MICRO SERIES

Jim Lyle

USB Primer
Classes and Drivers

Part 2 of 4

Now that we have some of the USB basics from Part 1, we're raring to go with USB! Jim wonders if an OS can provide all the drivers for the many devices there are today. With USB classes, he explains, it's entirely possible.

One of USB's earliest and most important goals was to make it easy to use. It has to be easy because the computer marketplace is rapidly expanding to include increasingly less-technical users.

These users don't know what an interrupt or DMA channel is, let alone how to finesse them into a working configuration. Nor should they have to. Even highly technical users are tiring of the difficulties involved in configuring or upgrading their computers.

From my perspective, it's not that difficult to install an ISA or PCI card. I've been doing this for years and I know how to set the jumpers (plug-and-play usually takes care of it anyway). I rarely get the cables on backwards anymore or offset by one row of pins, either.

But, one part of the process still strikes fear into my heart. One part of the installation never goes quite the way the instructions claim (when I finally do get around to reading them). There's one element that rarely fails to "blue screen" the machine repeatedly and strangely:

THE DRIVER

I've spent days trying to install the drivers for a seemingly simple device. Sometimes, it's incompatibilities with other drivers or software. Sometimes, the driver wasn't tested well or has a bug and needs a patch or upgrade. Sometimes I never do find the problem.

Wouldn't it be nice if all the drivers you ever needed came with the OS? You'd just plug something in and it would work. No more installation headaches; no problems moving from one machine to the next or even from one type of machine to another (e.g., from PC to Mac to Linux to workstation). There would be reduced disk and memory requirements, too, and one-stop shopping for upgrades. Overall, compatibility and reliability would improve dramatically.

Developers would find tremendous advantages as well, bringing more products to more platforms in less time and with less effort. Adding USB would no longer require the expertise (and the time, often in the critical path) needed to write drivers. Testing and support requirements would be reduced, and so would the overall project risk.

There are thousands of different kinds of devices already, and more are on the way. An OS can't possibly provide all the drivers for all of these types of devices. Or can it?

Although lots of different products are or will be available, many of them have more similarities than differences. In some cases, identical devices are produced by different manufacturers. In other cases, the products are different but the functions are similar.

Consider mice, track balls, and touchpads. They are physically different (and there's variation even within those broad categories), but the overall function is the same—moving a cursor on the screen. They all provide an *x* and *y* displacement and two or more buttons (or the equivalent).

What about full-page scanners, hand scanners, digital still cameras,

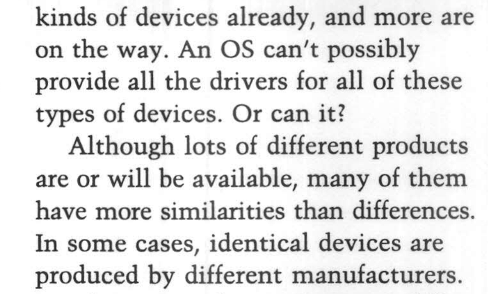


Figure 1—*USB uses well-defined protocol layers to reduce complexity and improve standardization.*

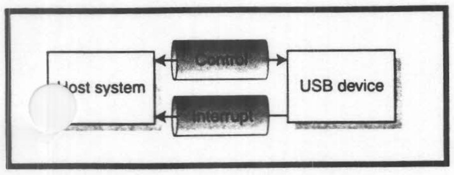


Figure 2—*USB devices use logical "pipes" to transfer information. This device uses two.*

and slow-scan video cameras? All produce an image of some form. Or printers? Color or black-and-white, laser or inkjet, Postscript or not—all put an image on paper.

With a little insight and forethought, most devices can be grouped into fewer categories, each with a common purpose and set of requirements. Then, it's possible to define a common API for each category and therefore the requirements of a single, generic driver suitable for use with any of the devices in that group. USB is trying to accomplish exactly this by defining a variety of device classes.

The USB specification defines the mechanical and electrical requirements for all USB devices as well as the fundamental protocols and mechanisms u to configure the device and transport data. The class definitions are add-on documents that refine the basic mechanisms and use them to establish the class-specific blueprint for both the device and the generic driver.

There will always be unique devices as well as manufacturers that choose to differentiate their product from the competition within the driver. For these cases, vendor-specific drivers will always be necessary.

But for most products, it'll be possible to use generic drivers that are part of (or included with) the OS. That's one of the most important advantages of USB.

Comm, Printer, Image, Mass Storage, Audio, and HID (human interface device) are a few of the defined USB classes. Some devices may fit into more than one category.

For example, there are combination printers/scanners. Although physically this is one device, logically it is two. Part of the device fits into the Printer ɩass and uses that generic driver. Part of it fits into the Image class and uses that driver. Devices in more than one class are called compound devices.

DEVICE CLASSES

Windows 98 includes many but not all of the USB class drivers. This situation is unfortunate, but it couldn't be helped because some of the class definitions weren't finished in time (some still aren't complete).

Future releases and service packs will add additional class drivers until most or all of them are available and supported. Apple and Sun Microsystems also have class driver implementations available or underway for their respective platforms.

As the name implies, HIDs are designed for some kind of human input or output. The most common examples are keyboards, pointer devices like mice, and game controller devices such as joysticks and gamepads.

This class also includes things like front panels or keypads (e.g., on a telephone or a VCR remote control), display panels or lights, as well as tactile and audible feedback mechanisms—essentially, anything you might press, twist, step on, measure, move, read, feel, or even hear.

Seemingly, this class would include almost anything connected to a computer, but it doesn't. Its primary purpose is control. Although it's very flexible, this class definition doesn't handle large amounts of data well. It doesn't need to; other device classes can better serve that purpose.

In a USB speaker, for example, the volume, tone, and other controls fall well within the HID class. But, the sound channels are data intensive, so they are better handled by the Audio class. In fact, many products in the other classes are compound devices with HID handling the controls.

Given the diversity of USB applications in general and HID devices in particular, how can any one driver hope to do all the things required by its class? The first part of the answer comes from the physical interface. There's only one! All USB devices communicate with the host via their USB port.

This sounds self-evident, but the implications are tremendous. The USB port works according to the same basic principles for all devices, in all modes of operation. The class driver never needs to worry or know about ISA or PCI buses, SCSI, IDE, or ATAPI interfaces, serial ports, parallel ports, keyboard ports, mouse ports, game ports, or anything else for that matter.

The class driver doesn't even need to know much about USB ports. Even that physical interface is abstracted and managed by the USB Host driver. This abstraction, or layering, is another key concept that makes class drivers possible.

Each layer has its own responsibilities and it uses APIs provided by the lower levels to accomplish them. It doesn't need to know how the lower levels work or which ones are present.

Figure 1 shows a simplified view of the various protocol layers that might be present for a USB device. Note that there are connections at all levels, but most of these are logical.

The single physical connection is between the USB host controller and the device's USB interface and is at the lowest level (shown in black). This layer is the hardware—the cables, connectors, and state machines.

The first layer of software, which is required in all cases (in light blue), is the USB host driver on the computer. On the USB device it is the essential firmware that manages the hardware and provides the standard requests (also called "chapter 9" requests because they are in that chapter of the specification). There's a logical connection between these layers for configuring and controlling the USB interface.

The device driver layer comes next (shown in grey) and is usually the class driver(s) on the host side and the corresponding firmware on the device side. The logical connection at this level carries class-specific commands and requests, although these often use protocols modeled after those in the layer below.

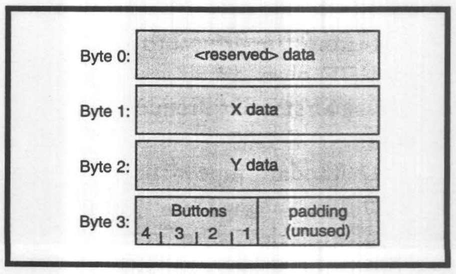


Figure 3—*This sample report for a USB joystick shows you one possible data organization.*

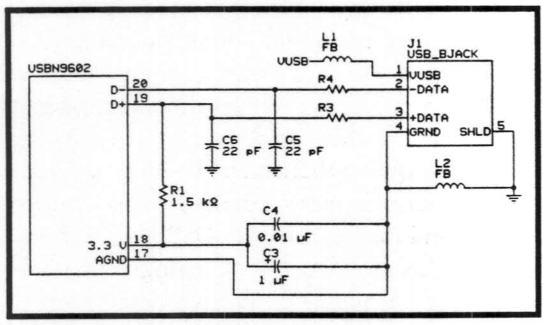


Figure 4—*This schematic shows you a typical connection between the USBN9602 and the USB connector (or cable).*

The top (dark blue) layer is the one the user sees and cares about. For example, the client software on the host might be a flight simulator and the associated function might be a joystick. At this layer, the only thing the client software (and user) cares about are the joystick inputs. It doesn't care (and doesn't need to know) how those inputs are read, packaged, and transported.

PROTOCOL LAYERS

To communicate with a USB device, the host software opens up a series of pipes, and uses them to transport data. The pipes correspond to hardware endpoints, which are individual channels, usually with dedicated buffers or FIFOs.

Pure HID devices use only two pipes (see Figure 2). The control (default) pipe, required by all USB devices, is used for receiving and responding to specific requests or commands. The standard requests use this pipe, and many of the class definitions (including HID) add class-specific requests.

The interrupt pipe sends asynchronous data to the host. This pipe is poorly named; USB doesn't support true interrupts but rather enables the device to predefine a maximum poll interval. This way, if a key is pressed, the mouse moved, or the joystick steered, the device can report in a timely fashion without a specific request (from the driver) to do so.

The HID class driver starts with the physical/standard request API common to all USB devices and adds the HID standard pipe structure and command superset. The difference from one HID device to another is the data it returns and what the data means.

HID data is packaged into structures called reports. Figure 3 shows a sample report for a joystick. It's simple and composed of four bytes.

The first byte is unused here but is reserved for a throttle position on another model. The second and third bytes are the *x* and *y* coordinates, respectively. The fourth byte contains information about the four buttons (one button per bit, with four unused bits that are zero-filled to pad out the byte).

This is just one example for one joystick. Other HID devices have different report structures. Other joysticks may have other structures, too. Some may order the data differently or have additional functions and capabilities (e.g., force-feedback).

SAMPLE REPORT

Obviously, the HID class driver can't keep report maps for all possible implementations of all possible devices. The device has to be able to describe the report to the class driver. This too is in keeping with standard USB mechanisms.

USB devices use predefined data structures called descriptors to describe their identification, capabilities, requirements, and protocols. The USB spec defines device and configuration descriptors that must be provided by all devices. The HID class definition adds information to these and goes on to define a report descriptor.

The report descriptor provides the map that the HID class driver needs to understand and interpret the report. The structure of the report descriptor is complex, though flexible. Fortunately, it doesn't complicate the device-side firmware because it is a data structure that can be written and compiled externally and then remain constant.

Listing 1 shows a sample report descriptor. The details are beyond the scope of this article, but note that it defines the type of application, size, maximum and minimum values, and subtypes of the various report fields.

Listing 1—*This `ReportDescriptor` function corresponds to Figure 1. USB devices use descriptors to describe themselves to the host PC.*

```
unsigned char ReportDescriptor[59] = {
 0x05, 0x01,       /* USAGE_PAGE (Generic Desktop) */
 0x15, 0x00,       /* LOGICAL_MINIMUM (0)          */
 0x09, 0x04,       /* USAGE (Joystick)             */
 0xa1, 0x01,       /* COLLECTION (Application)     */
 0x15, 0x00,       /* LOGICAL_MINIMUM (0)          */
 0x26, 0xff, 0x00, /* LOGICAL_MAXIMUM (255)        */
 0x75, 0x08,       /* REPORT_SIZE (8)              */
 0x95, 0x01,       /* REPORT_COUNT (1)             */
 0x81, 0x03,       /* INPUT (Cnst,Var,Abs)         */
 0x05, 0x01,       /* USAGE_PAGE (Generic Desktop) */
 0x09, 0x01,       /* USAGE (Pointer)              */
 0xa1, 0x00,       /* COLLECTION (Physical)        */
 0x09, 0x30,       /* USAGE (X)                    */
 0x09, 0x31,       /* USAGE (Y)                    */
 0x95, 0x02,       /* REPORT_COUNT (2)             */
 0x81, 0x02,       /* INPUT (Data,Var,Abs)         */
 0xc0,             /* END_COLLECTION               */
 0x15, 0x00,       /* LOGICAL_MINIMUM (0)          */
 0x25, 0x01,       /* LOGICAL_MAXIMUM (1)          */
 0x75, 0x01,       /* REPORT_SIZE (1)              */
 0x95, 0x04,       /* REPORT_COUNT (4)             */
 0x81, 0x03,       /* INPUT (Cnst,Var,Abs)         */
 0x05, 0x09,       /* USAGE_PAGE (Button)          */
 0x19, 0x01,       /* USAGE_MINIMUM (Button 1)     */
 0x29, 0x04,       /* USAGE_MAXIMUM (Button 4)     */
 0x55, 0x00,       /* UNIT_EXPONENT (0)            */
 0x65, 0x00,       /* UNIT (None)                  */
 0x95, 0x04,       /* REPORT_COUNT (4)             */
 0x81, 0x02,       /* INPUT (Data,Var,Abs)         */
 0xc0              /* END_COLLECTION               */
};
```

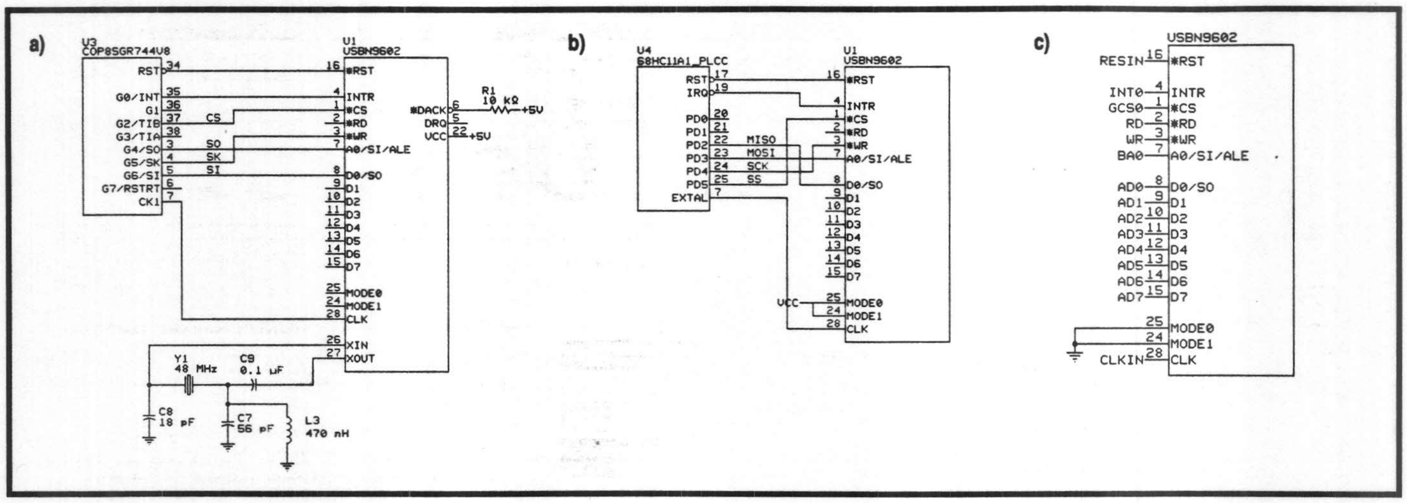


Figure 5—*This schematic shows a serial interface between the USBN9602 and a COP8 microcontroller. It also shows the oscillator circuit.* **b**—*Here's another serial interface. In this case, the microcontroller is a 68HC11.* **c**—*In this parallel interface to the USBN9602, the microcontroller is an Intel 80C188EB. But, this example would be typical of any case where an 8-bit data bus is available.*

So, a class-compliant USB product can entirely specify what it is and how it works in the onboard firmware. This makes the job of building, testing, and modifying a USB interface easier and more modular, and it brings it within the capabilities of most developers.

In the joystick, there are only three essential blocks—the ADC, USB interface, and microcontroller. The micro ties it all together, sampling the joystick at intervals and passing the data up through the USB interface (also managed by the micro).

The only new element is the USB interface. There are many varieties available: some are integrated with the microcontroller and some are separate components. These interfaces contain the state machines and buffers necessary to transmit and receive serial data on the USB. Conceptually, it's a smarter-than-average UART-like function.

National Semiconductor's USBN-9602 is one example of a USB interface. One side is attached to the USB cable or connector with a circuit like the one in Figure 4. (This figure and the ones following are not complete schematics; they merely highlight specific functions and interfaces.)

C3 and C4 bypass the USBN9602's internal voltage regulator (used by the internal USB transceiver). R1 is the required pullup that the device uses to signal its presence (and data rate) on the bus. The other components reduce EMI and transmission line effects to provide a cleaner signaling environment.

TYPICAL CONNECTIONS

The other side of the USBN9602 is the data path to the microcontroller. This data path is flexible and allows easy use with a variety of serial or parallel interfaces (there's even a DMA interface for high data rates).

Figure 5a shows a Microwire interface to a COP8 microcontroller, as well as the requisite dot clock oscillator circuit. Figure 5b shows an SPI interface to a 68HC11, and Figure 5c shows a parallel interface to an 80C188EB.

To make it even easier, several USB device manufacturers provide sample firmware source code. For the USBN-9602, National provides source code in C with compiler options for all of the microcontrollers mentioned here (and readily ported to others). This code is available on the web. Such firmware provides a ready-made solution to the some or all of the necessary device-side protocol layers.

If you want to build a mouse, keyboard, or other HID device, just modify the descriptor tables and a few top-level (function and class layer) firmware routines. Even if you're not building an HID device, the firmware layer that manages the USB interface device and responds to the standard requests provides a solid basis to start with.

PLAIN AND SIMPLE

USB simplifies the lives of developers and experimenters alike. It's possible for OSs like Windows 98 to provide most of the drivers you'll ever need for USB devices via class drivers, which make USB easier to incorporate into products and embedded systems. ■

Jim Lyle is a staff applications engineer at National Semiconductor where he has worked with flash memory, microcontrollers, and USB products. Jim has also worked as a development engineer and technical marketing engineer for Tandem Computers, Sun Microsystems, and Troubador Technologies. You may reach him at jim.lyle@nsc.com.

RESOURCES

USB information, www.usb.org
HID device information, www.usb.org/developers/hidpage.htm and www.microsoft.com/hwdev/hid
USBN9602 firmware source code, www.national.com/sw/USB

SOURCES

USBN9602, COP8
National Semiconductor
(408) 721-5000
Fax: (408) 739-9803
www.national.com

68HC11
Motorola
(512) 895-2649
Fax: (512) 895-1902
www.mcu.motsps.com

80C188EB
Intel Corp.
(602) 554-8080
Fax: (602) 554-7436
www.intel.com

MICRO SERIES

Glen Reuschling

USB Primer
Low-Speed USB Host Controller

Part 3 of 4

To wrap up our series on the Universal Serial Bus, Glen presents a two-part discussion on building USB hardware from scratch. This month, he lays the groundwork for putting together a USB host controller.

Almost two years ago, I read a trade-publication announcement for a new 8-bit USB microcontroller from Cypress Semiconductor. Coming from a background in PID and PLC controllers, I immediately saw the potential for such a chip.

At $1 apiece, you could create families of USB-smart motors, sensors, and actuators, all plugged into a USB-based microcontroller. Imagine not having to worry whether a thermocouple is J, K, or T type because the calibration and correction factors are already taken care of by the USB microcontroller.

A standard digital interface eliminates the need to configure special inputs and outputs before shipment to a particular customer. The tiered star arrangement of the USB interface lends itself nicely to the wiring structure of many industrial machines and eliminates the rat's nest of wires that converges at the back of an embedded multiloop controller.

The resulting collection of USB smart peripherals forms a simple distributed processing network, a potentially useful feature. The hot-swap and plug-and-play features of the USB offer the possibility of being able to do service and maintenance on a machine without costly shutdowns.

Not only does a digital signal path offer tremendous noise immunity over millivolt-level analog signals in the industrial environment, but it also allows for the use of optoisolators for electrical isolation—a must in an environment where 220 and 440 VAC are standard working voltages.

Finally, the ability to put a USB microcontroller to sleep means low power consumption for those situations where that's a consideration. With all of this in mind, I ordered the Cypress USB Development Kit ("USB Micro," *Circuit Cellar* 88).

When it arrived, I fired it up, wrote, downloaded, and ran a few simple programs, but that was it. My older workbench computer doesn't have USB ports, nor were there any USB interface plug-in cards available.

On top of that, the only USB software driver available was a beta version for Windows 95. All my development software is DOS- and Windows 3.1-based, and I wasn't ready to buy a new Pentium motherboard with USB support for my workbench and then install Windows 95 on it.

Not to be deterred, I looked into the availability of a chip or chipset that I could use to implement a USB host controller. The few I could find were all targeted to the PCI bus. None were targeted to the embedded microcontroller market.

I checked into the possibility of obtaining a USB host controller as intellectual property in HDL format, but price tags were five and six figures in size. Another dead end.

After downloading and reviewing the Universal Serial Bus Specification, Rev. 1.0, it was apparent that the low-speed USB specifications were a somewhat reduced and doable subset of the full-speed USB specifications. At this point, I decided that I could and would build my own low-speed USB host controller from scratch.

Eight months and many dead ends later, I finally put together the four-port USB host controller shown in Photo 1, tied it to an 8051-type microcontroller, and got it to talk to the CY7C3650 USB development card. So, what started out as a personal challenge to build my own USB host controller from scratch turned into a comprehensive USB learning experience.

This article recounts the work I did in building my low-speed USB host controller. By describing the problems I encountered and the solutions I came up with, my goal is to fill in the gaps in the standard USB documentation and make the USB more accessible.

This article is not a comprehensive review of the USB. I assume you're already familiar with the basics of the USB operation. For those of you who need to brush up on it, there are many good introductory articles to be found in various hardware-oriented publications, starting with Parts 1 and 2 of this *Circuit Cellar* MicroSeries.

Other excellent sources of information on the USB interface can be found in the datasheets and application notes published by the various chip manufacturers that offer USB products. A good example is Anchor Chips' web site, where a number of documents are posted that contain a wealth of tutorial-level information.

HOST CONTROLLER'S ROLE

Within the USB operation, the host controller has a distinguished role in contrast to those devices that are referred to as USB microcontrollers. The USB is a half-duplex serial bus with only one bus master (i.e., the host controller), but all of the USB microcontrollers that are commercially available are (by design) bus slaves and cannot perform the host-controller functions.

The advantage of the USB is that by offering a single standard interface for all low- and medium-speed peripherals, it eliminates the hassles of installation and configuration. It also offers true plug-and-play and hot-swap capabilities—features which, until now, have not been generally available to desktop PC users.

Although funneling all the different peripheral device communications through a single interface may make life easy for the PC user, it poses a major challenge to the hardware and software developer. The problem with this arrangement is that a single interface must now do the work of all the various plug-in cards and software device drivers that currently reside in the desktop PC.

Now, all the functions that could previously be spread out over a number of pieces of hardware and software must all be incorporated into a single combined hardware/software interface—the host controller.

To deal with the range of demands that the different kinds peripheral devices put on the USB interface, the USB Specification calls for four different modes of data transfer (control, interrupt, isochronous, and bulk transactions) and two different bus speeds (full and low speeds with data rates of 12 and 1.5 MBps, respectively).

Applications pass data through the USB by supplying the host controller with pointers to memory locations where data is to be moved from or to. In turn, the host controller keeps track of everything by using linked lists of linked lists of these pointers.

It is this complexity of the host-controller function (i.e., its need for fast memory bus access and a fast CPU to handle the computational overhead of each peripheral's associated device driver) that have restricted the commercially available full-speed USB host controllers to machines with either a PCI or other fast Local bus and OSs like Windows 98.

FULL SPEED VS. LOW SPEED

The full-speed USB protocol supports all four modes of data transfer with a maximum bandwidth of 1.5 MBps and a maximum data payload per transfer of 1023 bytes per packet per 1-ms frame for isochronous transactions [1, p. 55].

The host controller is required to prioritize and schedule all individual transfers, track the timing on isochronous transactions, and do error checking for bulk transactions. That's why full-speed USB host controller functions will always be beyond the capacity of the small embedded microcontroller.

On the other hand, low-speed USB supports only control and interrupt transactions, the two modes that entail the least computational overhead. Although the theoretical maximum bandwidth for low-speed transactions is 187.5 KBps, the maximum packet size per transaction is eight bytes.

This fact, combined with the maximum bus-access frequency per device of once every 10 frames, translates to a maximum data payload of only 800 Bps per low-speed endpoint channel (i.e., a doable data rate, even for a small 8-bit micro) [1, p. 57].

If you intend to interface exclusively to low-speed USB devices, a USB host controller becomes a possibility for small embedded micros. The requirements to support low-speed host controller functions are well within the capabilities of any fast 8-bit controller.

Regarding low-speed data rates, the bus access rate of once every 10 frames is a specification, not a hardware limit. It may be possible (depending on the low-speed device being used) to exceed this 800-Bps data rate by accessing it more often than once every 10 frames.

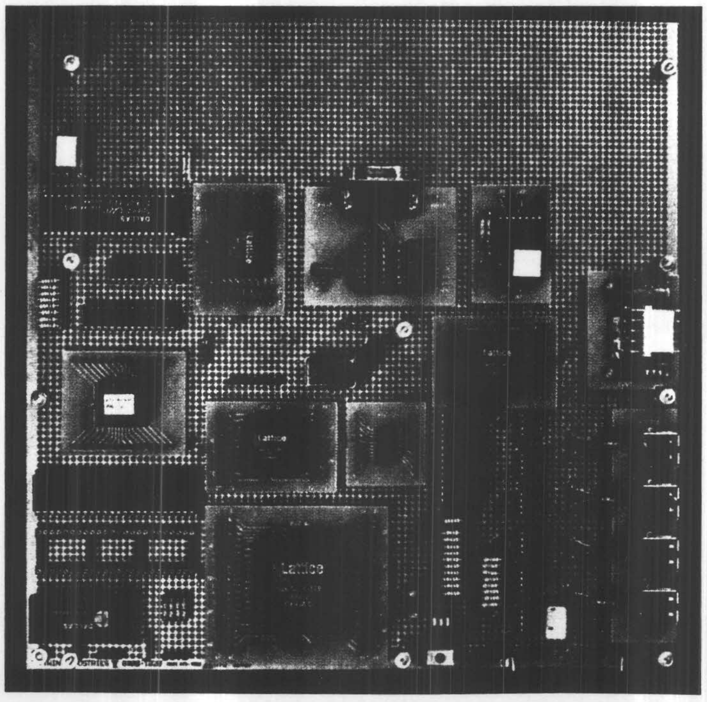
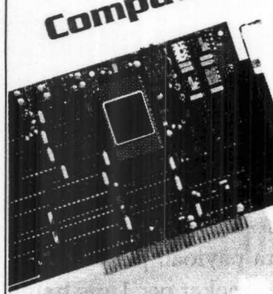


Photo 1—*This is it: the final version of my host controller project. The parts along the top and left of the photo include a standard 8051-type microcontroller, and the parts filling in the bottom right form the μSIE.*



LabVIEW^{tm} Compatible !

Lowest Cost

ADAC's **Value-Line** data acquisition cards - now with full-function LabVIEW VIs! Save $$ on new or existing LabVIEW applications. Download the VIs for free at **www.adac.com.**

Lowest Cost
5500MF **$195**
8 channels 12-bit A/D,
16 digital I/O, Counter/Timer

High Speed
5508LC **$245**
8 channels 12-bit A/D,
100KHz, DMA

Multi-I/O, DMA
5516DMA **$295**
16 channels 12-bit A/D,
DMA, 16 digital I/O

High Resolution
5500HR **$495**
16 channels 16-bit A/D,
DMA, 8 digital I/O

High Performance Data Acquisition Since 1974

voice	**800-648-6589**
fax	**781-938-6553**
web	**www.adac.com**
email	**info@adac.com**

ADAC
American Data Acquisition Corporation
70 Tower Office Park, Woburn, MA 01801 USA

LabVIEW is a trademark of National Instruments Corp.

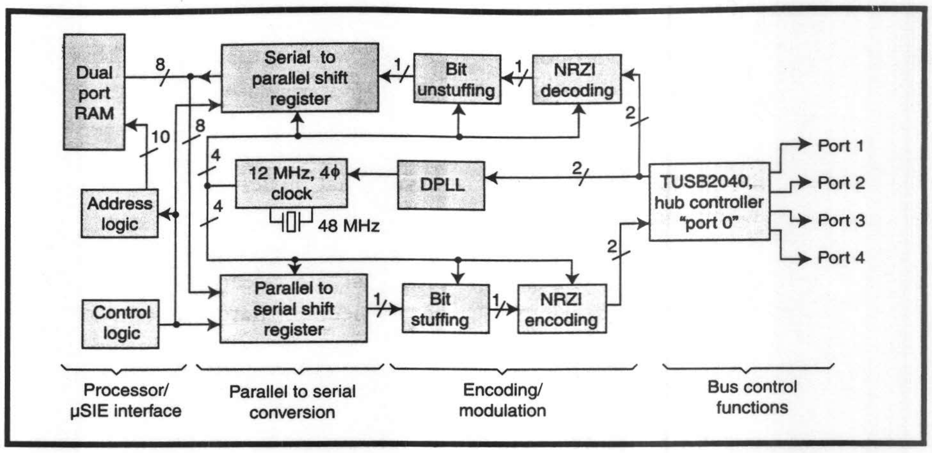


Figure 1—*These are the basic functional building blocks that compose my low-speed host controller's µSIE.*

A second way to multiply data rates to an USB device is to access multiple endpoints within a single device. However, low-speed devices are limited to only two endpoints besides the default endpoint 0 [1, p. 47].

On the mechanical side, full-speed USB specifies a shielded twisted-pair cable for the signal lines, but for low speed, neither twisted pair nor shielding is specified [1, p. 86]. Maximum specified lengths are 5 m for full-speed and 3 m for low-speed cables [1, p. 89].

But, for practical purposes, signal integrity and the bus turnaround time set the upper usable length limits. So, despite the specifications, with proper cable selection, you should be able to wire a house-sized area with low-speed USB peripherals.

No doubt you've already encountered the term serial interface engine (SIE). The SIE is to the USB what the UART is to the RS-232 interface. The only functional difference between the two (from the processor's point of view) is that a UART passes data on a FIFO buffer (possibly only one byte deep), while the SIE passes data directly to and from the processor's memory via pointers.

For the sake of discussion, I'll use the term "micro serial interface engine" (µSIE) to refer to the minimum hardware necessary to implement completely and correctly all low-speed USB host-controller functions. Interestingly enough, the µSIE hardware requirements for the host controller are less than those for a USB slave SIE.

Because the USB protocol encompasses both hardware and software issues, it makes sense that a host controller is partly hardware and partly software in its makeup. For this reason, there are many different ways to build a host controller, depending on which functions are implemented in hardware and which ones are implemented in software.

HARDWARE INTRODUCTION

The first step in building a USB host controller is to visit the USB Implementers Forum web page and download:

- "Universal Serial Bus Specification," Rev.1.0 and the white papers
- "Cyclic Redundancy Checks in USB"
- "Designing a Robust USB Serial Interface Engine (SIE)"

For a membership fee of $2500 per year, you can join the USB Implementers Forum and have extended access to documents and tech support. But, this membership fee is out of the range of a lot of people, including me.

If you choose to download and read the USB Specification, two notes of warning are in order. First, the full USB specifications encompass issues of software, hardware, and communications protocol.

Unfortunately, the authors of the document use the same words at different points to reference alternately hardware, software, and communications protocol issues. Although this overloading of definitions may not bother a C++ programmer, it blurs the lines between the different aspects of the USB specifications and makes them confusing to read from a hardware builder's point of view.

Second, the USB Specification is n and was never written to be a s ematic guide for building USB hardware. With patience, most of the information you need to guide you in building a USB interface can be found in the USB Specification. As for details not explicitly stated in the Specification, be prepared to resort to some trial and error work to find them.

To help you locate information within the USB Specification and elsewhere, I included a list of references that source specific page numbers and sections.

HOST CONTROLLER HARDWARE

Figure 1 shows the minimum hardware configuration necessary to implement a USB host controller. A short guided tour of this basic µSIE starts with the output section, which is the easiest to implement.

USB uses 8-bit bytes as its basic unit of data. So, as in all serial interfaces, the signal path begins with a parallel to serial conversion. Data is loaded byte at a time and shifted out starting with the least significant bit.

The next step in the signal path is encoding and modulation, which includes bit stuffing and NRZI encoding [1, pp. 121–122]. Extra bits are added to the signal stream to ensure adequate transitions for syncing and clock separation. Then, data information is encoded as a differential signal and control information is encoded as DC level signals [1, p. 115].

The serial signal stream is then passed to the output buffer section. This section is responsible for line driving, slew rate control, hot-swap power management functions, and low- and full-speed device detect.

The return signal path becomes more complicated. Because there's no separate clock line in the USB, the receive clock must be derived from the incoming signal. This function is performed by a digital phase lock loop (DPLL) [2, p. 2].

DC level control signals must be etected, bus time-out intervals must be monitored, and this information passed on to the control section of the µSIE. Once it is past this input section, the return serial signal path is just the reverse of the outgoing path.

To avoid using memory pointers, the µSIE uses a dual-port RAM as the processor/µSIE interface. Rather than passing pointers, the microcontroller's software is responsible for placing data to be sent or received only at specific memory locations. This is an example of a hardware/software tradeoff you can make when building a host controller.

Although the control logic is shown as a small block in Figure 1, it represents a major portion of the µSIE. Also, not shown is the 1-ms frame clock and the functions of CRC generation and checking that can be implemented quite easily in software by the microcontroller.

JUST GETTING STARTED

Now you've got a USB background and some resources to check out for more information. Next month, I'll introduce my low-speed USB host controller and the lessons I learned while putting it all together. My goal is to show you that there's no reason USB should remain only the province of desktop PCs. ■

Glen Reuschling is a manufacturing engineer by day and a serious hobbyist by night. His most recent home project resulted in this article. You may reach him at wildiris@cruzio.com.

REFERENCES

[1] USB Implementers Forum, *Universal Serial Bus Specification*, Rev. 1.0, www.usb.org, 1996.

[2] USB Implementers Forum, *Designing a Robust USB Serial Interface Engine (SIE)*, www.usb.org.

SOURCES

USB Development Kit
Cypress Semiconductor
(408) 943-2600
Fax: (408) 943-6848
www.cypress.com

Anchor Chips, Inc.
(619) 613-7900
Fax: (619) 676-6896
www.anchorchips.com

MICRO SERIES

Glen Reuschling

USB Primer

Building a USB Host Controller

Part 4 of 4

If you're looking for detail, you have no excuses left: USB is a mystery no longer. By describing the step-by-step construction of his low-speed USB host controller, Glen reveals the inner workings of USB hardware.

Last month, I introduced my host controller. Now, I want to show you how it all came together.

First, a note on the choice of parts that went into it. I chose the TI TUSB-2040 four-port USB HUB for the core of the µSIE because it's a stand-alone part that requires no programming and is available in a DIP package.

HUB support chips were chosen based on the application examples in the TUSB2040 datasheet. CPLDs, which support in-circuit programmability, are a must for the initial prototyping of your host controller.

As we go through the process, you may want to have these files (available on the Circuit Cellar web site) handy:

- `HUB01C.PLD`—programming for low- and full-speed clock generation, the DPLL, and bus time-out detect logic
- `HUB04C.PLD`—dual-port RAM address control logic
- `HUB02C.PLD`—µSIE functions

To make signal tracing easier, the net names used in Figures 1 and 2 match the variable names used in the `.PLD` files.

A simple way to implement subroutines within state-machine logic is to use a multiphase clock. One level of state machine controls the clock signals to the next level down.

The top-level state machine controls the idle, start of frame, and send/receive states. At the next level down are the state machines that generate send and receive transactions. The third and fourth levels are responsible for bit and byte manipulation.

Although the complete µSIE (including the dual-port RAM) can be put into a single, large CPLD, it's less expensive to implement a design using several smaller devices. But always pick a bigger part than you think you need. To help debugging, leave a few extra pins on your CPLD for test points.

GETTING STARTED

Start with a development card or prototyping system for your favorite 8-/16-bit microcontroller. A debugger for downloading and running code is essential. And be prepared to generate a number of different test routines.

Set aside some memory space for a dual-port RAM; 1 or 2 KB is enough. This RAM will be the main driving engine for your host controller. A good storage oscilloscope or logic analyzer is a must for debugging the hardware.

CLOCKING THE µSIE

Several system clock signals must be generated for the µSIE [3]: a four-phase full-speed 12-MHz clock, a four-phase 1.5-MHz low-speed clock, and the 1-ms frame period clock. The µSIE uses a 48-MHz crystal oscillator as the primary source for all timing, so generating the 12- and 1.5-MHz four-phase clock signals becomes straightforward.

The USB Specifications call for the 1-ms frame clock to be adjustable [1, p. 124], but this is from the perspective of a USB slave device. Many USB slave devices use ceramic resonators instead of crystals for their clock so some provision was included to compensate for their lower accuracy and stability.

The 1-ms frame clock is implemented as a divide-by-n counter, with n giving the adjustability called for [4]. Because the host controller is the bus master and sets the 1-ms frame period, it doesn't need to be software adjustable. So n is fixed equal to 48,000.

Regarding the 1-ms frame clock as implemented in `HUB01C.PLD`, the more switching the CPLD does, the more power it dissipates. At 48 MHz, the Lattice part gets quite warm, so the divide-by-*n* counter was broken up into several stages, each being clocked by a slower signal from the stage before.

FEEDING THE µSIE BABY

The first test routine to run on your 8-/16-bit micro is one that, upon a hardware interrupt generated by the host's 1-ms frame clock, writes a few bytes of data out to a starting location in the dual-port RAM. Make the first byte (i.e., `ByteCount`) the number of bytes in your test packet.

The first HDL state machine to download to the CPLD is one that, when initiated by the 1-ms frame clock, starts by reading the number of bytes to transfer and loads the data sequentially from the dual-port RAM into a shift register [5] and clocks it out as a serial stream. Output clocking is a shift-right least-significant-bit-first operation.

Exactly how the data is read from or written to the dual-port RAM depends on what stage of the build-and-test process you're at. Understanding the final form of the dual-port RAM's read/write logic must wait until after the inner workings of the µSIE have been explained.

Using the serial bitstream generated by the first test routine, start implementing in HDL code the bit-stuffing and NRZI-encoding functions [1, pp. 121–122]. Bit stuffing is accomplished by counting the number of 1s (in a row) sent and, when that count hits six, clocking a 0 into the NRZI encoding section without clocking the serial-out shift register [6].

USB is a two-wire medium, so there can be up to four signal states. Data information is encoded as a differential signal; control information is encoded as a DC-level signal [1, p.115].

The USB uses three of the four possible states—J logic state, K logic states, and the single-ended zero (SE0) state used for sending control information [7]. Also, the NRZI state machine needs a disconnect or floating-output state.

The Sync Pattern determines the initial state and initial transition for the NRZI state machine. When the host controller initiates a transaction, it takes the bus from a floating state to the Idle state.

The first Sync Pattern transition is from Idle to K. So, the NRZI state machine must start in the disconnect state, go to the J state for one bus-clock period, then transition to the K state. After this startup, the NRZI encoder runs itself.

In NRZI encoding, data is encoded as transitions, not levels. A 0 gives a transition; no transition indicates a 1. When viewing the USB signal on your logic analyzer, don't look at it as a sequence of 1s and 0s. The transitions between 1s and 0s represent the bits in the USB serial signal.

DIFFERENTIAL LOGIC LEVELS

The differential logic levels for low-speed USB communication are the reverse of those of the full speed, but the SE0 control signal is still the same. Reversing prevents full-speed devices from responding to low-speed signals.

But, this reversing is only from the perspective of the slave controller. The host still sends out all transactions in full-speed differential logic levels.

The last hub device before the receiving low-speed device is responsible for reversing the sense of the differential signal levels [1, p. 224]. So, the µSIE doesn't need to implement or support the full-/low-speed signal-level reversal.

Now that your prototype host controller can generate a serial bitstream of the right form, issues of USB protocol must be dealt with (i.e., what bytes of data and in what order must they be written to the dual-port RAM).

Due to definition overload, the information in chapter 8 of the USB Specification, "Protocol Layer," can be difficult to interpret. Unfortunately, this chapter is essential for USB hardware design.

Figure 3, which shows a Start Of Frame (SOF) packet and a complete USB hub Control Read sequence, may help elucidate the hardware-level details of the USB transaction protocol. It's an expanded version of the Control Read sequence in Figures 8–12 of the Specification [1, p. 154].

The Control Read sequence shown here is the `GET DESCRIPTOR` device request. Because the USB uses the duration of the SE0 state to encode control signals, time-delay information is critical.

The next USB test signal the host controller needs to generate is the SOF packet [1, p. 149]. So,

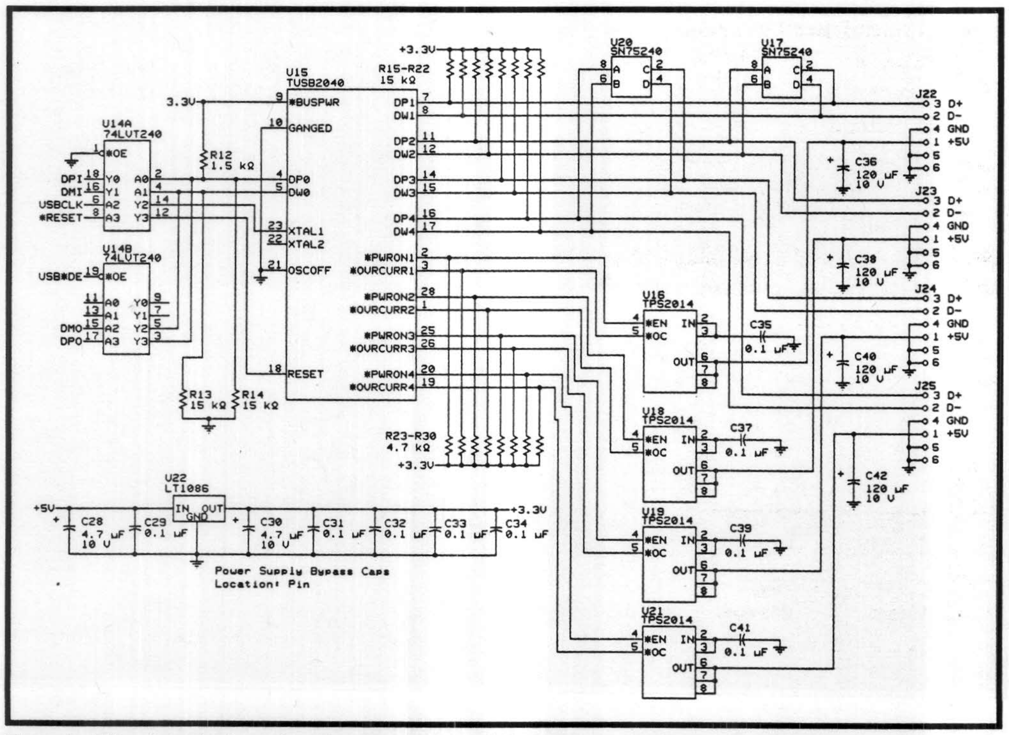


Figure 1—*This schematic shows the wiring diagram for the TUSB2040 four-port USB hub. The TUSB2040 is a +3.3-V device, so a buffer (U14) was included to translate between +3.3- and +5-V logic levels.*

the second test routine to download and run on your 8-/16-bit controller is one that, upon a hardware interrupt generated by the host's 1-ms frame clock, increments an 11-bit frame counter, constructs the next SOF packet, and writes that packet to your starting location in the dual-port RAM.

I let the microcontroller generate all data fields and the µSIE generate all control signals. My controller does Sync Pattern, PID, Frame Count, CRC5, CRC16, and ACK; the µSIE generates the Idle and End of Packet (EOP) states.

Sync Pattern is seven 0s followed by a 1—that is, the byte 80h shifts out least significant bit first [1, p. 123]. When this byte is run through bit stuffing and NRZI encoding, it becomes [Floating, Idle,K,J,K,J,K,J,K,K] in USB form or [HiZ,1,0,1,0,1,0,1,0,0] as logic levels.

The 5-bit CRC is easily generated via a 2-KB look-up table, with the 11-bit frame count as the address offset [8].

In addition to the bytes fed to it via the dual-port RAM, the host needs to append the final EOP and Idle control states. The EOP is a SE0 state, and the Idle is a J state. After these are sent, the USB signal lines are floated in anticipation of a possible return signal.

The EOP and Idle control states are specified as time durations, which are different for low and full speeds [1, p. 125]. It's easier to think of them in terms of bus clock periods, the EOP being two and the Idle state being one bus clock period, respectively.

For USB, the word "transaction" refers to the process of data transfer between the host controller and the slave USB device. It's not a hardware-level concept but a protocol- or software-level concept. At the hardware level, the basic communication unit is the packet.

A transaction consists of a sequence of packets (initiated by the host) being sent back and forth between the host controller and a USB slave device. The host's µSIE sends and receives individual packets. Completing a full transaction is a software issue for the 8-/16-bit microcontroller.

The USB Specification calls for ten different packet types, each one being identified by its PID and representing a different handshake message or data message type [1, p. 146]. A USB transaction starts with the host controller sending a Token packet, or, in the case of low-speed transactions, a Special packet followed by the Token packet.

The Token packet initiates communications, but after the initial Token packet is sent, there may or may not be any further packets sent or received. For example, the SOF transaction consists of only the SOF Token packet.

Here's a classic example of the definition overloading that makes reading the USB specification difficult for the uninitiated. Depending on the context, "SOF" can refer to a transaction type, a packet type, or a Token PID.

Building the SOF packet starts with the microcontroller placing the `ByteCount` byte, then the sync byte, the PID byte, and two bytes for the frame count and CRC5 into the dual-port RAM. For example, with a frame count of 54h, the SOF packet passed to your µSIE is [04h,80h,A5h,54h,90h].

The µSIE input state machine reads in and shifts out these four bytes of data, then goes to the SE0 state for two bus clock periods, then to the Idle or J state for one period, and then disconnects [9].

Congratulations! You've just completed your first USB transaction.

THE USB HUB

The next step is a full-speed USB hub Control Read sequence. The '2040 hub controller chip was enlisted to do all of the bus housekeeping functions, including line driving, slew rate control, hot-swap power management functions, and low-/full-speed device detect. This is a nonstandard use for this part, but it does the job and saves work in recreating (as separate circuitry) all of the functions it performs.

The '2040 is a USB slave controller and, before its functions can be accessed, it must be initialized just as any other USB device. All USB hubs are full-speed devices, so the clock-generating section contains logic for a 12-MHz full-speed clock signal.

Fortunately, it's only necessary to talk to the '2040 hub to initialize it. Listening isn't required at this stage, so the first USB communication that the host controller will initiate is a full-speed Control Read sequence with its own '2040 hub.

Note that in wiring the '2040 device, because control information is sent as a DC-level signal, it's crucial that you pay attention to the correct pull-up and pull-down resistors in your design.

The Control Read sequence consists of a number of transactions, each consisting of the exchange of two or three packets. For example, the setup transaction consists of a SETUP token packet followed by a DATA0 data packet and ending with an ACK handshake packet.

At this point, a logic analyzer or storage oscilloscope becomes a necessity to monitor the serial signals going to and coming from the '2040. Monitoring ensures you're getting the right signals out to it, in the right order, and with the right timing, and lets you verify that the Control Read sequence is completing correctly.

The first packet in the first transaction of the Control Read sequence is a SETUP packet. The SETUP token is interpreted by the hub the same as an OUT token.

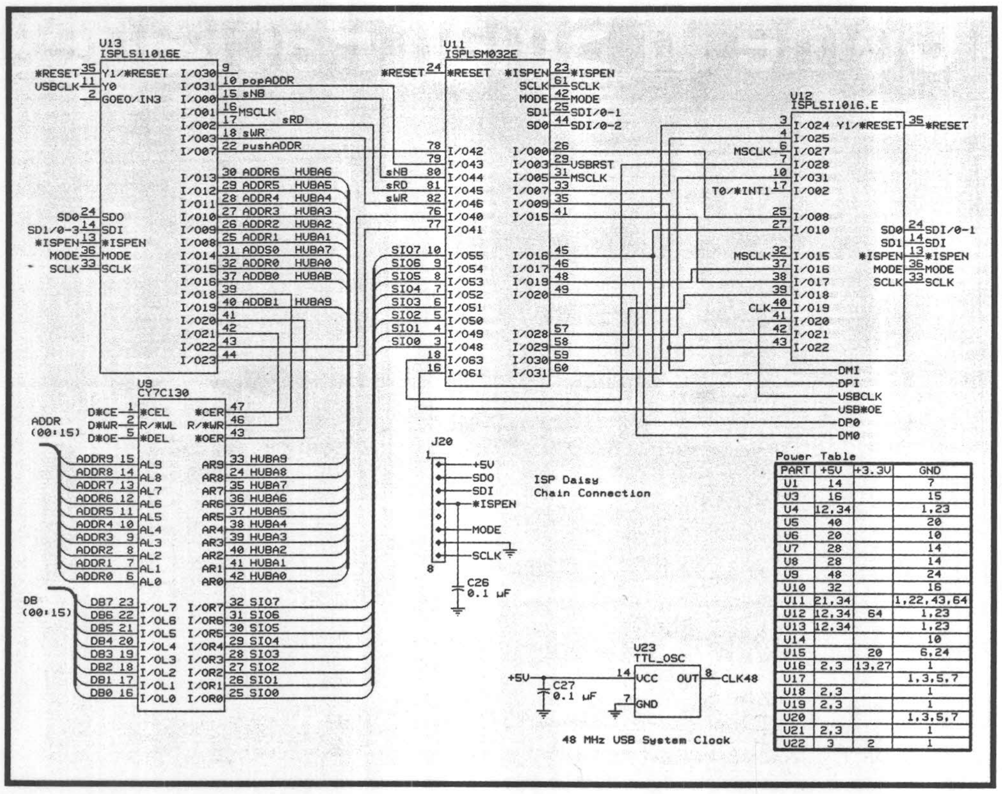
On powerup, all USB devices default to address 00h and endpoint 00h. These will be the initial ADDR and ENDP values for the TUSB2040 part.

To get the CRC5 value for this or any SETUP, IN, or OUT packet, combine the ADDR and ENDP values to form an 11-bit address. Then use this value to offset into the CRC5 look-up table [10].

As with the SOF packet, building the SETUP packet starts with the microcontroller placing the `ByteCount` byte, then the sync and PID bytes, and two bytes for the ADDR, ENDP, and CRC5 into the dual-port RAM. Your result should be [04h,80h,2Dh,00h,10h].

The purpose of this SETUP packet is to send a message to endpoint 0 of the device at address 0 to expect incoming data on the next packet out. But after getting this message packet, the receiving device will only listen for so long before timing out, which brings up the issue of bus turnaround time.

The bus turnaround time is basically the time period (after the end of a packet's transmission) that any USB device still expecting to receive another packet will wait before "hanging up." The USB Specification calls for at least 16 (but no more than 18) bus clock periods for this interval [1, p. 161].



Power Table

PART	+5V	+3.3V	GND
U1	14		7
U3	16		15
U4	12,34		1,23
U5	40		20
U6	20		10
U7	28		14
U8	28		14
U9	48		24
U10	32		16
U11	21,34		1,22,43,64
U12	12,34	64	1,23
U13	12,34		1,23
U14			10
U15		20	6,24
U16	2,3	13,27	1
U17			1,3,5,7
U18	2,3		1
U19	2,3		1
U20			1,3,5,7
U21	2,3		1
U22	3	2	1

Figure 2—*This schematic shows the host-controller support logic. The nets ADDR[00:15] and DB[00:07], left side of the dual-port RAM, connect to your 8-/16-bit microcontroller. Programming for the three CPLD devices is available via ftp.*

The second packet in the Setup transaction is a DATA0 packet. DATA0 and DATA1 are the two packet identifiers used to distinguish data packets. The first data packet in any transaction is always labeled with the DATA0 PID.

The second data packet begins with the DATA1 PID, and the third with the DATA0 PID. They then alternate between DATA1 and DATA0, except for Control sequences, which end with the exchange of an empty DATA1 packet.

BACK TO THE µSIE

So far, the dual-port RAM read/write logic has only had to output one packet at a time. Because of the short bus turnaround time allowed for full-speed USB devices, your DATA0 packet must be ready to go right after the SETUP packet is sent, which means it must already be in the dual-port RAM along with its associated SETUP token packet.

That's just one example of the general case, that, given the short bus turnaround time, the host controller must be able to complete a full transaction all at one time.

On the microcontroller side of the dual-port RAM, transactions are stored in a semi-linked-list fashion (i.e., the `ByteCount` byte, which starts each data sequence, is used as an offset to the next `ByteCount` byte). If this count is zero, the SIE knows that no more data is to be sent and goes to an idle state until the beginning of the next 1-ms frame clock.

To keep the 8-/16-bit microcontroller from causing conflicts by trying to access the same segment of the dual-port RAM at the same time as the µSIE, its memory is divided up into four 256-byte pages [11]. Each page is in turn divided into 128-byte output and 128-byte input segments [12].

The four memory pages form a cyclic queue, with the micro (via its I/O pins) keeping track of the current page the µSIE is on. While the µSIE is accessing the current page, the micro is writing to the output segment of the next page and reading from the input segment of the previous page.

Along with these modifications to the dual-port RAM address control logic, changes have to be made to the µSIE programming. Two changes are the additions of an "expect return flag" and a "data ready flag" [13]. Because the value of `ByteCount` for low-speed transactions never exceeds 16, the lower four bits of this byte are used for `ByteCount` and the upper four bits are used for control flag information [14].

With this modification, your µSIE will know from the `ByteCount` byte that the SETUP packet expects no return handshake and that there's a second packet waiting in the dual-port RAM to be sent out before the bus turnaround time expires. This is probably the biggest single leap in your HDL code development, so expect to spend some time working through it.

INITIALIZING THE USB HUB

This DATA0 packet contains the USB device request information necessary to start the hub initialization process [1, chap. 9]. All USB devices must be initialized by the following minimum exchange of information.

On startup, all USB devices default to ADDR 0 so the host issues a `GET DESCRIPTOR` device request to the default control endpoint, ENDP 0, at ADDR 0. The device address is set to a new value and the device descriptor is requested again at the new address.

The `GET DESCRIPTOR` request consists of [1, p. 176]:

- byte 1—bmRequestType, 80h, device request
- byte 2—bRequest, 06h, request code
- byte 3—wValue, 00h, descriptor index
- byte 4—wValue, 01h, descriptor type
- byte 5—wIndex, 00h, not used
- byte 6—wIndex, 00h
- byte 7—wLength, 12h, 18 bytes is standard descriptor length
- byte 8—wLength, 00h

As was the case for the 5-bit CRC, the microcontroller is responsible for generating the CRC data. A 16-bit CRC is generated using a simple algorithm combined with a look-up table [15]. Make sure that the two CRC16 bytes are placed in the right order in the dual-port RAM to ensure that the resultant serial data clocks out correctly.

The Setup transaction expects a handshake after the DATA0 packet is and an Out transaction. The next step is to write a test routine for your 8-/16-bit micro to complete each transaction (one per 1-ms frame period).

THE FIRST IN TRANSACTION

The first In transaction consists of an IN packet sent by the host controller, followed by a DATA1 packet from the hub. It ends with the host controller returning an ACK handshake packet.

So far, the µSIE can only send packets; it can't receive them. Because clock, the DPLL skips the fourth phase clock state. This could pose a problem for any µSIE implementation that, like mine, uses this fourth phase clock signal.

There are ways around this bug, but I haven't spent the time to fix it yet. So far, it hasn't presented any problems because the hub runs off the same 48-MHz clock as the µSIE, and low-speed transactions don't last long enough (in bit times) for this skipping of clock states to occur.

need to be detected.

A two-bit shift register helps accomplish this detection. As the USB signal is clocked in, a comparison is made between the current and previous signal states. If they are equal, a 1 is clocked out; if not, a 0 [19].

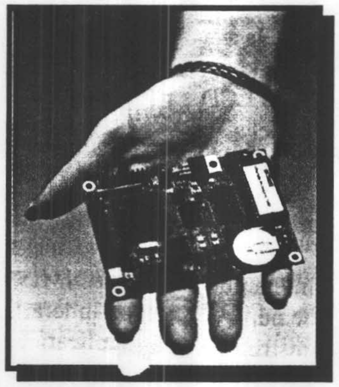
Bit unstuffing is just the reverse of bit stuffing. If the number of 1s in a row equals six, the input NRZI decoding state machine is clocked once without clocking the input shift register [20]. This, in effect, throws away

MEASURE, ANALYZE, CONTROL COMPUTER

FAST FACTS

- 8 Channel, 10-bit A/D
- 2 PWM, analog outputs
- Pulse timing/measurement
- 19 Digital I/O lines
- 512K Battery backed RAM
- LCD Port
- Y2K compatible clock
- 2 RS-232 serial ports
- 32K Flash EPROM
- Low power modes
- Debug software included
- From $99 in 100's

The RPC-220 is used for control and data acquisition.



Applications:
- Data logger
- Remote measurement and control
- Wireless network

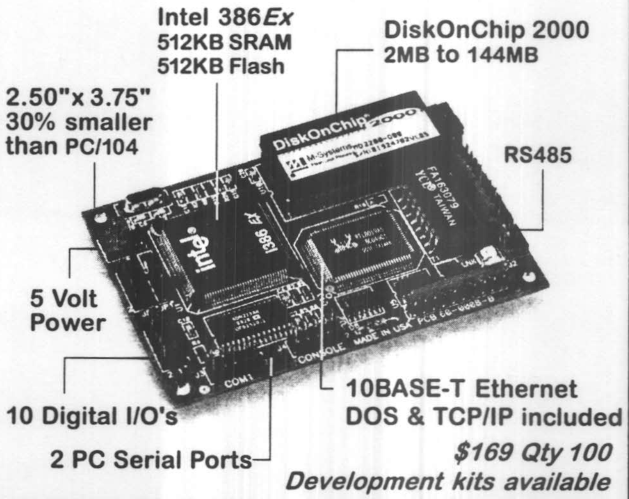
Use any of the application programs included with the Development System as a basis for your own.

Call now for more information
800-642-9971
or 303-690-1588 www.rp3.com

REMOTE ® PROCESSING

Fax: 303-690-1875 info@rp3.com
7975 E. Harvard Ave., Denver, CO 80231

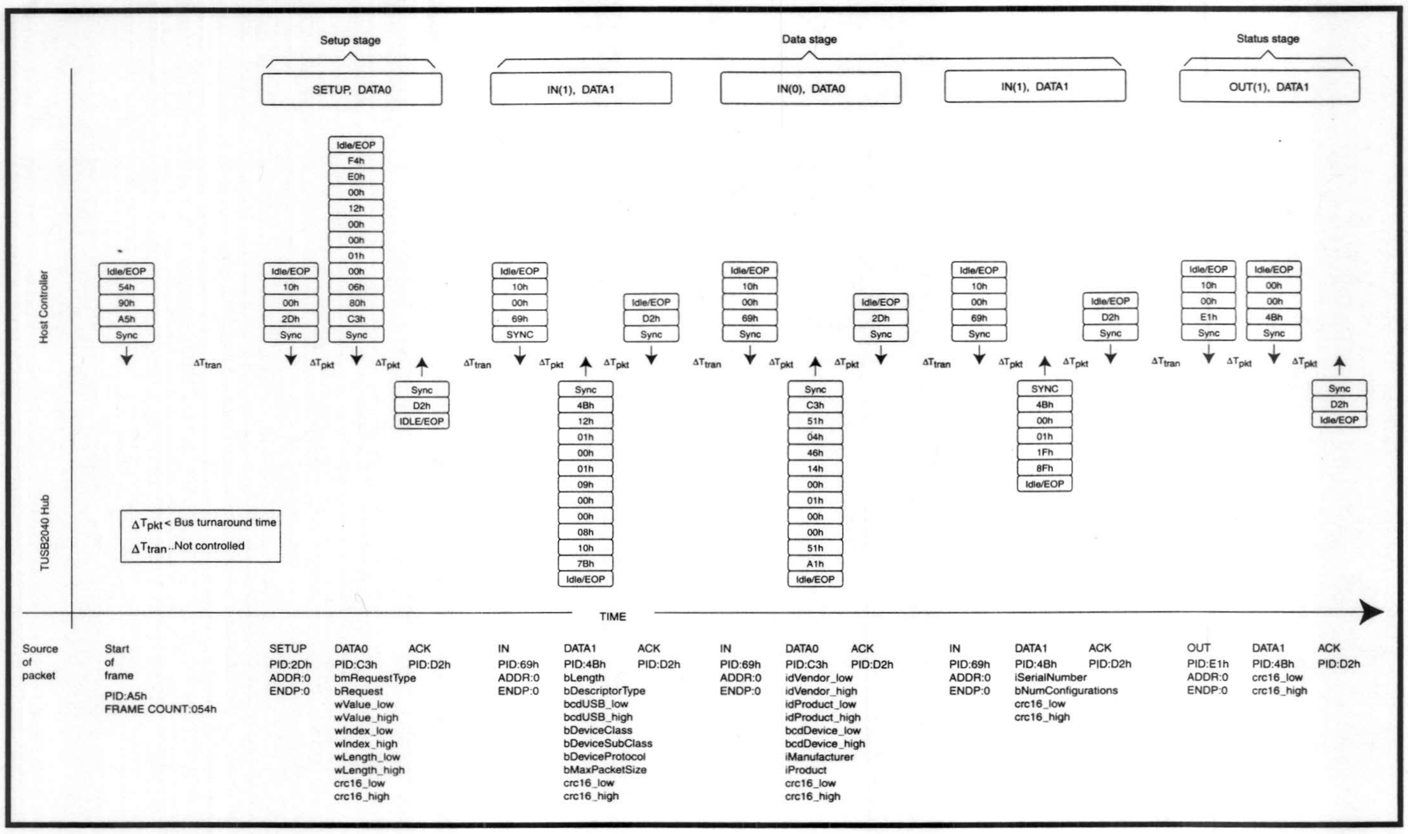
Give Your Product Ethernet & TCP/IP



$169 Qty 100
Development kits available

Call 530-297-6073 Fax 530-297-6074
For more information see our web site at
www.jkmicro.com/uflash

JK *microsystems*



Setup stage
SETUP, DATA0
Data stage
IN(1), DATA1
IN(0), DATA0
IN(1), DATA1
Status stage
OUT(1), DATA1
Host Controller
TUSB2040 Hub
Idle/EOP 54h 90h A5h Sync
Idle/EOP 10h 00h 2Dh Sync
Idle/EOP F4h E0h 00h 12h 00h 00h 01h 00h 06h 80h C3h Sync
Sync D2h IDLE/EOP
Idle/EOP 10h 00h 69h SYNC
Sync 4Bh 12h 01h 00h 01h 09h 00h 00h 08h 10h 7Bh Idle/EOP
Idle/EOP D2h Sync
Idle/EOP 10h 00h 69h Sync
Sync C3h 51h 04h 46h 14h 00h 01h 00h 00h 51h A1h Idle/EOP
Idle/EOP 2Dh Sync
Idle/EOP 10h 00h 69h Sync
SYNC 4Bh 00h 01h 1Fh 8Fh Idle/EOP
Idle/EOP D2h Sync
Idle/EOP 10h 00h E1h Sync
Idle/EOP 00h 00h 4Bh Sync
Sync D2h Idle/EOP
ΔTtran
ΔTpkt
ΔTpkt < Bus turnaround time
ΔTtran ..Not controlled
TIME
Source of packet
Start of frame PID:A5h FRAME COUNT:054h
SETUP PID:2Dh ADDR:0 ENDP:0
DATA0 PID:C3h bmRequestType bRequest wValue_low wValue_high wIndex_low wIndex_high wLength_low wLength_high crc16_low crc16_high
ACK PID:D2h
IN PID:69h ADDR:0 ENDP:0
DATA1 PID:4Bh bLength bDescriptorType bcdUSB_low bcdUSB_high bDeviceClass bDeviceSubClass bDeviceProtocol bMaxPacketSize crc16_low crc16_high
ACK PID:D2h
IN PID:69h ADDR:0 ENDP:0
DATA0 PID:C3h idVendor_low idVendor_high idProduct_low idProduct_high bcdDevice_low bcdDevice_high iManufacturer iProduct crc16_low crc16_high
ACK PID:D2h
IN PID:69h ADDR:0 ENDP:0
DATA1 PID:4Bh iSerialNumber bNumConfigurations crc16_low crc16_high
ACK PID:D2h
OUT PID:E1h ADDR:0 ENDP:0
DATA1 PID:4Bh crc16_low crc16_high
ACK PID:D2h

Figure 3—*Here you see an SOF packet for frame count 054h, followed by the Control Read sequence for the TUSB2040's* `GET DESCRIPTOR` *device request. The actual USB signal is formed by shifting out, least significant bit first, in the order they appear and in NRZI fashion, the bytes shown in this diagram.*

the stuffed bit. Just a reminder here: input clocking is a shift-right least-significant-bit-first operation.

After you have a decoded signal, you need to start looking for the Sync Pattern. Separating it from the data signal and detecting its end will enable the µSIE to lock to the correct timing for reading data bytes out of the serial input shift register.

The scheme I decided upon was to look for the value 8h in the upper four bits of the input shift register [21]. But, you can't start looking too soon because the first few bits to come through the decoding process may be interpreted as a 1 preceded by 0s.

The first segment of the hub's descriptor comes as the following DATA1 packet of Sync Pattern, PID, data and crc16: [80h,4Bh,12h,01h,00h, 01h,09h,00h,00h,08h,10h,7Bh]. If this is the data output you see from the '2040 (via your µSIE), then your input section is working correctly.

The EOP SE0 state is detected using a counter clocked in the 4× domain. For example, with 12-MHz full-speed transactions, the counter is clocked at 48 MHz [22].

The counter is gated by the detection of a SE0 state. If the counter detects two full bus clock periods, the `EOP_flag` is set [23]. In my implementation, the ending Idle state is not detected.

Because a host controller isn't allowed to return either a NACK or STALL handshake, the µSIE doesn't have to implement these functions [1, p. 151]. The only transaction responses the host controller can return are an ACK handshake for transactions that are completed correctly and a no-handshake return for transactions that don't complete correctly.

The simplest choice is for the host controller to acknowledge everything and let the microcontroller take care of error checking and repeating a transaction if and when errors occur. Since I chose this option, the handshake packet need not originate with the µSIE but can be generated by the 8-/16-bit micro and loaded into the dual-port RAM with all the other packets in a transaction. An ACK packet is just the SYNC and PID bytes: [02h,80h,D2h].

To finish the In transaction, the host controller must issue an ACK handshake. If the EOP for the DATA1 packet has been detected successfully, the µSIE reads in the next packet to send from the dual-port RAM (i.e., the ACK packet.

The final transaction of any Control Transfer is the Status Stage, which is nothing more than the exchange of an empty DATA1 packet. Basically, it gives the sending party a chance to send a parting ACK or NACK response back to the receiving end [1, p. 155].

BURPING THE µSIE BABY

If all has gone right so far, you now have a µSIE correctly locking on to the incoming USB signal and outputting data bytes in a coherent fashion from its serial-in parallel-out shift register. The next step is to get this data into the dual-port RAM in a way that is readable by the 8-/16-bit micro.

There are two address counters in the dual-port RAM address logic—one for reading output data, one for writing input data. When the µSIE goes into receive mode, it captures the current write address but doesn't write to it. It then clocks the write address counter.

As the µSIE inputs data and writes it out to the dual-port RAM, it keeps a count of the total number of bytes are Control No-Data transactions. Setting this configuration value to the nonzero constant corresponding to its hub configuration gives the host access the '2040's hub controller functions.

Once these functions are accessed, the host controller, using an Interrupt In transaction, can read the Port Status Change bit map at the '2040 hub's endpoint 1 to see if any devices are attached [1, p. 262]. A GET PORT STATUS request returns specific information about a port's attached device.

The SET PORT FEATURE request is then issued several times with different wValues until the hub and device are ready. For example, wValue 04h is reset, wValue 01h is port enable, wValue 08h is power on, and so on [1, p. 254].

Configuring the USB hub is an example of bus enumeration [1, p. 169]. At startup, all USB devices require the same initialization and configuration sequences and share almost all of the same standard device requests.

When it detects the presence of a new device on the USB, the host controller enables an initial 100 mA of current to bus-powered devices. Next, there is a wait period for the attached device to power up and be ready to accept communications [1, p. 242]. This period is specified by bPwrOn2PwrGood, which is returned as part of a hub's Hub Descriptor [1, p. 250].

A RESET device request is sent to the hub controller the device is attached to. The hub then issues a standard USB RESET control signal, which is at least 10 ms of the SE0 state [1, p. 119].

Following a reset, the first device request issued is always GET DESCRIPTOR at ADDR 0 and ENDP 0. Then, you have access to all device requests specific to that configuration class. USB devices may support more than one configuration, so you must specify a particular configuration value during device initialization.

The final step is to enable full power to those devices requiring more than the initial 100 mA of current.

BACK TO USB SCHOOL

Not every device supports every USB device request. There's a lot of information in chapters 8, 9, and 11 of the USB Specification that may be essential for communications with your USB device: for example, variables like bInterval (the interval period, in frame counts, between device accesses) or bPwrOn2PwrGood (the time for a device's power supply to ramp up).

Experiment with the standard device requests by issuing them to the '2040. Having feedback from an actual USB device and comparing its responses to the Specification's text will help you make sense of the material.

Low-speed transactions are prefaced by a special PID (PRE) sent at full speed. The µSIE doesn't append the usual EOP and Idle states when sending this low-speed preamble, but (after a pause of four bus cycle periods to give downstream hubs time to reconfigure) goes from the PRE PID to the Sync Pattern of the next low-speed packet [1, p. 160].

ON YOUR OWN

Now, plug a low-speed USB device into the host controller, power up, enable the device's features, and start teaching the attached micro to speak USB. Once the host controller is up and running, read the '2040's Port Status

What those features are and how you talk to the USB device after configuration is device-specific [1, p. 34]. So, from here on out, you're on your own.

I hope you'll take up the USB mantle, create your own embeddable low-speed USB host controller, and make your work available as HDL shareware. Happy USB'ing. ■

Glen Reuschling is a manufacturing engineer by day and a serious hobbyist by night. His most recent home project resulted in this article. You may reach him at wildiris@cruzio.com.

SOFTWARE

Programming for the three CPLDs was done using the hardware description language CUPL. Source code is on the *Circuit Cellar* web site.

REFERENCES

[1] USB Implementers Forum, *Universal Serial Bus Specification*, Rev.1.0, www.usb.org, 1996.
[2] USB Implementers Forum, *Designing a Robust USB Serial Interface Engine (SIE)*, www.usb.org.
[3–23] see REFS.TXT on the Circuit Cellar web site.

SOURCES

ispLSI1016E, ispLSI1032E
Lattice Semiconductor Corp.
(503) 681-0118
Fax: (503) 681-3037
www.latticesemi.com

TUSB2040
Texas Instruments, Inc.
(972) 644-5580
Fax: (972) 480-7800
www.ti.com